Hardware-based performance monitoring with *VTune Performance Analyzer* under Linux

Hassan Shojania
shojania@ieee.org

Abstract

All new modern processors have hardware support for monitoring processor performance. In this project, we try to explore use of VTune Performance Analyzer *for hardware-based performance monitoring of a Linux cluster of Pentium 4 Xeon processors.*

1 Introduction

Performance measurement of any high-performance cluster system is very critical for development and deployment of efficient applications for such system. All new modern processors have special hardware to monitor processor performance. This hardware-based performance measurement has many advantageous over traditional intrusive methods of performance measurements based on adding code for probing execution time of portion of a program. For example, data collected by this hardware provides performance information on applications, the operating system, and the processor. These data can guide performance improvement efforts by helping programmers tuning the algorithms used, and the code sequences that implement those algorithms [1].

In this project, we intend to explore *VTune Performance Analyzer* set from Intel for performance measurement of standard MPI/OpenMP-based benchmarks under our cluster of Dell PowerEdge 2650/6650 (2/4 way SMP systems with Xeon processors) running Linux. As the first step towards *VTune*, we are targeting mainly the proper system setup/configuration with limited trials of few benchmarks. This should provide enough background for more in depth analysis of other benchmarks.

This report is organized as follows: First we overview the basics of performance monitoring hardware in Section 2. Features of Pentium 4 performance monitoring hardware is overviewed in Section 3. In Section 4 we describe performance data collection in VTune with the help of concepts presented in Sections 2 and 3. Section 5 explains different deployment choices of VTune and installation steps. Section 6 provides some sample test results. In Section 7 we present our conclusions.

Section 2 and 3 have heavily used [1] and [2]. These are excellent sources for more information about performance monitoring hardware and features of Pentium 4 in this area. Of course, [3] provides raw register-level information for programming Pentium 4 performance monitoring hardware.

2 Basics of performance monitoring hardware

There can be different approaches for collecting processor performance data.

1. Modifying the application to add instrumentation code for collecting various data like instruction trace and memory reference data. This requires either rebuilding it from source code or modifying its executable version; both not favorable usually (especially for operating system code). Also, these approaches can disturb the applications behavior, bringing questions about validity of the collected data.

2. Another way to collect processor performance data is by using a simulator to model the processor as it executes the application. This simulation approach can yield a detailed data on processor blocks like pipeline stalls, branch prediction, cache performance, and so on. However, processor manufacturers do not usually provide simulators for advanced processor designs and third parties don't know enough about the hardware detail to build such a simulator.

3. Using performance-monitoring hardware has several distinct advantages over previous approaches. Having the processor itself actually collect performance data as it executes an application have several benefits. First, the application and operating system remain largely unmodified. Second, the accuracy of the collected event counts is much higher compared to using loose simulators which are not capable of simulating exact hardware behavior. Third, performance-monitoring hardware collects data on the fly as the application executes, avoiding the slow simulation-based approaches. Fourth, this approach can collect

data for both the application and the operating system. These advantages often make hardware performance monitoring the preferred, and sometimes only choice for collecting processor performance data.

Performance-monitoring hardware typically has two components: performance event detectors and event counters. Users can configure performance event detectors to detect anyone of several performance events (for example, cache misses or branch mispredictions). Often, event detectors have an event mask field that allows further qualification of the event. For example, based on processors privilege mode (user/supervisor) to separate events generated by application from operating system code, or for filtering accesses to the L2 cache based on cache line's specific state (i.e. modified, shared, exclusive, or invalid).

Further configuration is usually possible through enabling event counters only under certain edge and threshold conditions. The edge detection feature is most often used for events that detect the presence or absence of certain conditions every cycle, like a pipeline stall. The threshold feature lets the event counter compare the value it reports each cycle to a threshold value and then increment the counter. The threshold feature is only useful for performance events that report values greater than one in each cycle, for example for an "Instructions Completed" event, number of cycles when three or more instructions were completed (in one cycle) can be counted by using a threshold of two.

2.1 Performance event monitoring

Performance events can be grouped into five categories: program characterization, memory accesses, pipeline stalls, branch prediction, and resource utilization. Program characterization events show largely processor-independent attributes of a program like number and type of instructions (for example, loads, stores, floating point, branches, and so on) completed by the program. Memory access events aid performance analysis of the processor's memory hierarchy, like references and misses to various caches and transactions on the processor memory bus. Pipeline stall event information helps users analyze how well the program's instructions flow through the pipeline. Branch prediction events show performance of branch prediction hardware. Resource utilization events can monitor the usage of certain resources like number of cycles spent using a floating-point divider.

2.2 Performance Profiles

Though hardware performance counters reveal many information about the software, but this information is very low level and mainly expose global state of the processor not a particular application behavior. And as long as the source of monitored behavior is not detected, the user can not improve the hardware/software performance. Two common approaches are:

Time-based Sampling (TBS) approach tries to capture the percentage of time an application spends in its different sections through exposing the most frequently executed portions of the code. It is usually implemented through interrupting an application's execution at regular time intervals and recording the program counter. At the end of the application, a histogram will show the number of samples collected for each section of code.

Event-based Sampling(EBS) produces a histogram of performance event counts based on code location. Now the application is interrupted after a specific number of performance events (a counter reaching some threshold) rather than at regular time intervals in TBS. Similar to a time-based profile indication of most frequently executed code locations, an event-based profile indicates the most frequently executed code locations that cause a particular performance event. This requires performance-monitoring hardware support for generating performance monitor interrupt when a performance event counter overflows. Then the Interrupt Service Routine (ISR) handler captures sample data from the program (e.g. program counter) and re-enables the interrupt for another interrupt.

2.3 Limitations

Mainstream processors suffer from a common set of problems:

Too few counters. Limited number of counters restricts monitoring multiple events concurrently.

Speculative counts. Some processors can't distinguish between performance events for instructions that do not complete (speculative execution) from the one who really complete. This is important because performance events are still generated (like cache misses) even for speculatively executed instructions that never retire.

Sampling delay. The sampling of program counter when a hardware event counter overflows (commonly used with EBS) can not identify the exact instruction that caused the overflow. This degree of accuracy is not required all the time (e.g. the associated module or thread is of more interest) but there are cases when an event frequency needs to be related to particular type of instruction. The program counter is sampled by a performance monitor interrupt (PMI) after overflow of an event counter. However, the current instruction

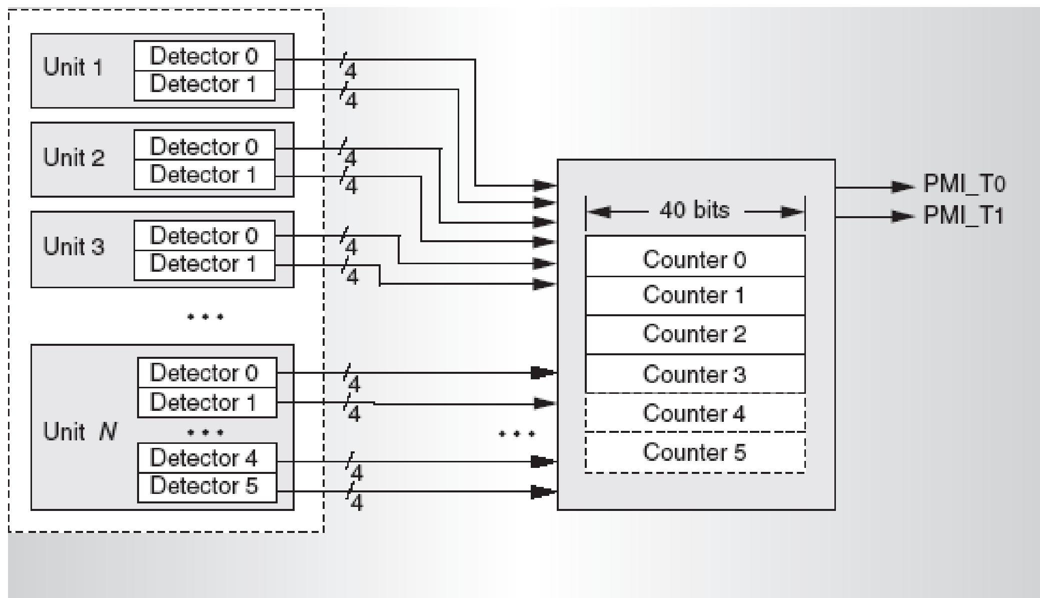

Figure 1: The general structure of the Pentium 4 event counnters and detectors) (from [2])

is usually not the instruction causing the overflow as processors pipeline cause an arbitrary delay between counter overflow and signaling of an interrupt.

Lack of data-address profiling. Usually there is no support for collecting profiles of *data addresses* generated by CPU when a memory hierarchy performance event happens.

3 Pentium 4 Performance-monitoring Features

Intel Pentium 4 has tried to overcome the limitations described in the previous section.

3.1 Increased event counters

Pentium 4 supports 48 event detectors and 18 event counters, allowing concurrent collection of a larger set of data (see Figure 1). To decrease the amount of signal routing among event detectors and save with silicon area required to associate each event detector to its own counter, several blocks of counters were dispersed across the chip to be *shared* by geographically close event detectors as Figure 1 shows. The event detectors select an event and mask it based on privilege mode or thread ID. Event counters support threshold comparison and edge detection (introduced in section 2). These are configured through event select control register (ESCRs) and counter configuration control registers (CCCRs). Figure A and B shows the format of these registers. For detailed description of them refer to Chapter 15 and Appendix A of [3]. Figures 2 and 3 show the general structure of event counters and detectors.

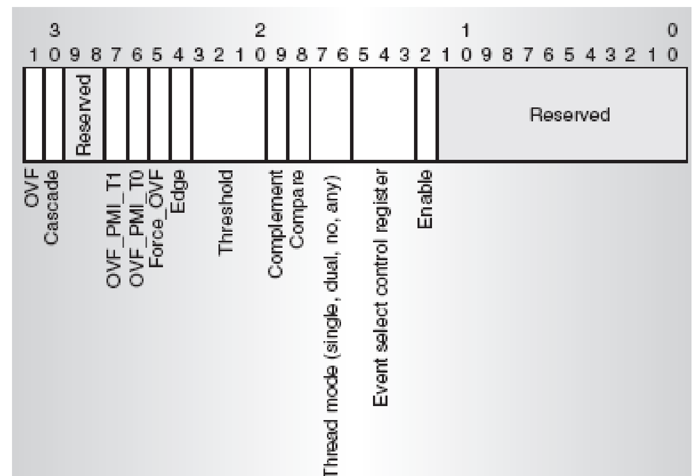

Figure 2: Counter configuration control register (CCCR) (from [2])

As the result, the 18 performance counters are grouped into nine pairs of counters. Each performance counter is associated with a fixed (CCCR) register (18 in total) to set up counter for a specific method or style of counting. Each one of 45 events can be monitored by associating the proper ESCR (45 in total) to a performance counter to keep track of the event count. Note that since only 18 counters are available and each hardware block is associated with some fixed set of performance counters, a total of 18 events with maximum of 4 or 6 events from each block (depend-

ing on the block) can be monitored at any time.

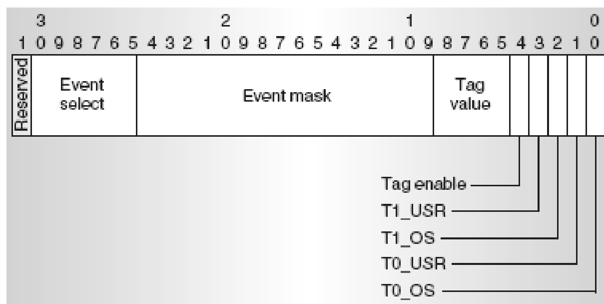

Figure 3: Event select control register (ESCR) (from [2])

Performance counters can be accessed through RDPMC (Read Performance Monitoring Counters) instruction. Execution of this instruction is restricted to supervisor mode only unless PCE (Performance-Monitoring Counter Enable) bit of CR4 (a control register) is enabled by a privileged code. CCCR and ESCR registers are accessed through RDMSR (Read from Model Specific Register) and WRMSR (Write to Model Specific Register) which both are restricted to code running in supervisor mode. As the result, all performance measurement tools (e.g. VTune, PerfCtr) have a driver portion running in kernel mode to program these restricted registers. For more information about these registers see section 15.9 of [3].

3.2 Instruction tagging

Pentium 4's instruction tagging mechanism allows distinguishing speculative from nonspeculative performance events. This is done through tagging the μops when they cause performance events. These tags are kept till μops are retired where they contribute to the proper event counter. Three provided tagging mechanism are:

Front-end tagging for events in early stages of the pipeline due to instruction fetch, instruction type and μop delivery from trace cache.

Execution tagging for some class of μops when they write their results back to register file.

Replay tagging for μops that are replayed because of conditions like cache misses, branch mispredictions, dependence violations and resource conflicts. See section 15.9.7 of [3] for more information.

3.3 Precise event-based sampling (PEBS)

Beside supporting imprecise event-based sampling (IEBS), Pentium 4 has added support for precise event-based sampling (PEBS) to allow identifying the exact instruction causing a performance event and also generating data address profiles. It uses a microassist (a mechanism typically used for handling infrequent and problematic conditions) to change next executed μop to a microassist service routine for capturing sample data. The use of microassist and its service routine rather than relying on interrupts (which are delayed because of pipeline depth), avoids the inaccuracy caused by traditional IEBS.

Since no longer an ISR executed at event overflows, microassist copies the sample data to a buffer (already programmed) in memory. A high watermark is kept by microassist to monitor buffer consumption after adding every new sample. If it is reached, then an ISR is generated to empty PEBS buffer. This helps to improve the overhead of event-based sampling as ISR is generated far less often than previous case when an ISR was fired for every single overflow of event counter.

3.4 Thread based qualification

Further, event detection can be qualified against the thread ID of the CPU thread executing the current code when simultaneous multithreading (SMT) is active. This can also be mixed with privilege mode. For example application level events on thread 0 or operating system events on thread 1. Note that here the thread ID means CPU thread number (e.g. zero or one on currently available two-level hyperthreaded Pentium 4s) not thread ID associate by operating system to each task's execution thread.

3.5 Current limitations

One of the main existing limitations is lack of support for "attributing event counts to specific task in execution". This is because of the **global nature of hardware performance counters** as they count events for all tasks in execution (time shared) on the processor. So currently there is no alternative except changing operating system to provide a local view of performance counters for each application (or its threads). This is not easy to implement. For example, PerfCtr ([4]) is providing such a task-based local view (only process level but not thread-based) of performance counters under Linux by hooking into kernel scheduler. But this is not implemented under Windows (e.g. by Intel VTune) as hooking into kernel schedular is not trivial in Windows.

The other problem is **microarchitecture dependence** of performance monitoring hardware. The definition of events, supported capabilities and software interface for programming the counters are all microarchitecure dependent. This makes it difficult to develop and maintain software tools to expose hardware performance monitor features across different microarchitectures. A few who have this support are: Intel VTune ([5]), University of Tennessee PAPI ([6]) and PerfCtr ([4]).

4 VTune Performance Analyzers

VTune Performance Analyzers software set developed by Intel collects and displays software performance data to help with identifying and locating performance bottlenecks in codes. Here are a brief overview of its features:

Supported processors: Intel Pentium 4, Intel Xeon, Intel Itanium and Itanium 2 processors, Mobile Intel Pentium III Processor M, Intel Pentium M Processor, Intel PXA255 processor and Intel PXA262 processor.

Supported operating systems: Both Windows and Linux are supported. The latest retail versions are: VTune Performance Analyzer 7.0 (for Windows) and VTune Performance Analyzer 1.1 for Linux.

Beta versions of next revisions currently available are: VTune Performance Analyzer 7.1 (for Windows) and VTune Performance Analyzer 2.0 for Linux. They are expected to be released early in 2004.

Low overhead profiling: VTune uses low intrusion system-wide sampling methods to provide most accurate representation of application's performance.

Source-level Tuning Advice: It employs Intel Tuning Assistant to examine application interaction with the system and provides both coding pitfall and processor-specific advice to help streamlining the code.

Multi-Threading/Multi-Processor Support: It allows viewing sampling data or call graph data for large numbers of threads simultaneously or isolating specific threads or processors (including Hyperthreaded processors).

Collecting Performance Data from Remote Systems: It allows configuration, start and stop of a remote computer's profiling data collection session from a host computer. Host computer must be a Windows system while the target system can be either Linux or Windows.

Scripting Capability: It allows writing scripts that automatically run VTune analyzer call graph or sampling activities.

Pack and Go Feature: It provides the ability to pack the project (including all collected data) from a Windows or Linux system and unpack it to a Windows system to allow graphical viewing of the Windows/Linux performance data.

VTune support three main performance collection mechanisms through its **data collectors**:

4.1 Sampling Collector

This method uses non-intrusive, instruction-address sampling collector to collect, analyze, and display system-wide software performance data. It can identify the critical processes, threads, modules, functions, and lines of code running on the system by analyzing the collected data. During sampling, the analyzer monitors all the software executing on system including the operating system, JIT-compiled Java* applications, .NET* applications, 16-bit applications, 32-bit applications, and device drivers. The VTune Analyzer analyzes the code associated with the collected samples and displays the hotspots or bottlenecks in the Hotspot view. User can drill down from the hotspots to the source or assembly code. Since the VTune Analyzer's sampling is non-intrusive and does not modify binary files, application performance is not impacted that much. There are two types of sampling mechanisms can be chosen from to collect data:

Time-based sampling (TBS) collects samples of active instruction addresses at regular time-based intervals (1ms. by default).

Event-based sampling (EBS) collects samples of active instruction addresses after a specified number of processor events.

Both were already reviewed in Section 2 earlier. When a Sampling Activity is started, the VTune Analyzer starts collecting samples by interrupting the processor at the specified sampling interval (TBS case) or specified number of detected hardware events (EBS case) and collects samples of instruction addresses. For every interrupt one sample is recorded, storing the execution context of the software (module, process, thread) currently executing on system.

4.2 Call Graph Collector

This method collects information about the program flow of an application, that is, how many times a function calls some other function and the amount of time each function spent by executing its code and/or calling other functions. A function can be a caller or/and a callee. In many cases, the caller may call the callee from several places (sites), so call graph also provides call information per site. It is possible to drill down from the function summary, graph, and the call list to the source and see call graph data summary by function.

Contrary to the Sampling collector, Call Graph collector uses instrumentation to modify the program so that dynamic information is recorded during program execution. Data collection routines invoked at specific points in the execution of the target program record run-time information. These routines provide information about time spent

in each function, and the call sequence that leads to a specific function. This process does not change the functionality of the program. However, it slows performance down. By default, the VTune instruments all application functions and system-level exports. The VTune analyzer keeps track of the exit and entry points, records the number of times each function was called, establishes a relationship between the caller (parent) and callee (child) function, and stores this data.

4.3 Counter Monitor Collector (only on Windows)

This method is to identify system level performance issues. Counter monitor selectively polls system performance counters, which are grouped categorically into performance objects. These counters are *system* performance counter exposed by the operating system and mainly doesn't related to a hardware performance counter. Some examples are: Non-paged pool allocations, Context-switches of a thread/sec and File data operations/sec. The data collected through this feature can be correlated with data collected by other collectors, such as sampling. Each sample is made by a *trigger* (e.g. at predetermined intervals according). Also, VTune can provide a *runtime view* of system performance counters by generating a graph that shows the counter changes as they happen.

4.4 Extending the features of VTune

VTune has a Software Development Kit (SDK) to assist developers in writing DLLs (for Windows) that extend the functionality of the VTune analyzer. This can be in a form of new performance counters in hardware devices, device drivers, or software applications that can be tracked in VTune. Also, the trigger model in the VTune enables developers to develop their own triggering mechanism for the Counter Monitor collector to collect performance counter data. This is done by calling some APIs in the application to notify VTune about the trigger.

In both Linux and Windows, the developer can target specific sections in the application for sampling by inserting calls to the VTPauseSampling() and VTResumeSampling() APIs in his code. To do that, the application must be likned with libVtuneApi.so (for Linux) or vtuneapi.lib (for Windows).

5 VTune setup for our Cluster

As mentioned earlier in this report, our main goal was to explore VTune's capabilities for performance measurement of MPI and OpenMP applications under our cluster of Dell PowerEdge 2650s/6650s running Linux. Also we wanted to be able to categorize performance data based on executing threads to analyze OpenMP applications in particular as this was not possible with our previous tool (PerfCtr).

5.1 Standalone setup

We first deployed *VTune Performance Analyzer 2.0 Beta for Linux*. This tool was installed on a standalone Linux system (i.e. no need to a *controlling system*) in the default path of */opt/intel/vtune*. The installed components consists of a driver portion installed in *[VTune-path]/analyzer/vtune_drv-[version info].o* and an application portion installed in *[VTune-path]/shared/bin/vtl*. As Section 3 already pointed, programming of hardware performance counters in Pentium 4 must be done in supervisor mode (e.g. by a kernel-mode driver). This is done by driver portion of VTune but since RedHat 9.0 is not a supported platform for VTune, the driver binary for this platform is not coming with the original installation. So a new driver must be built from the provided source code (*[VTune-path]/analyzer/drivers/src*). Since the driver depends on system kernel code, the ***exact*** source code of Linux kernel running in the system must be available. The path to kernel source code can be configured through *[VTune-path]/analyzer/drivers/src/configure* script. The built driver will have a name like vtune_drv-[version info].o in our case *vtune_drv-2.4.22smp.o* showing the kernel version and whether the kernel is uniprocessor (up) or multiprocessor (smp). Then the driver must be loaded by *[VTune-path]/analyzer/drivers/src/insmod-vtune* script. The driver must be reloaded after each reboot manually unless automatic load of driver was setup at installation of VTune.

Now by running the application portion *[VTune-path]/shared/bin/vtl*, different monitoring schemes can be configured through command line parameters or script.

A few notes:

- Installation must be done by the *root* user.
- The license file must be copied to the proper Intel license path (by default */opt/intel/licenses*; can override it by INTEL_LICENSE_FILE environment variable).
- Installation process also installs *EntireX DCOM for Linux* in */opt/sag* so *DCOM* service must be running before launching *vtl*(*DCOM* will launch */opt/sag/exx/v611/bin/ntd* apparently).
- *vtl* depends on some libraries in */opt/sag/exx/v611/lib*, so make sure this path is added to your *LD_LIBRARY_PATH* environment variable or included in */etc/ld.so.conf*. This was not done automatically by installation package.
- Make sure proper users are added to the *vtune* user group when installation asks about it. Otherwise, users might have problem running *vtl*.

5.2 Remote data collection

Though running the standalone version of VTune Analyzer works well, configuring and setup of data collection and also analysis of collected data are not very easy through command line setup or writing scripts. Pentium 4 has many events and configuration mask that a GUI based interface will speed up the collection setup. So we also explored using a remote data collection setup with the help of a Windows system running *VTune Performance Analyzer 7.1 Beta (for Windows)* as the controlling node. Through the GUI of Windows-based VTune the Linux node can be easily controlled and collected performance data is displayed in easily understandable charts and graphs. Also, the data can be exported to a .csv (comma separated values) file and later converted to an *Excel* spreadsheet.

We briefly overview the installation process here as it is more complicated than standalone case. After download of the package and unpacking it (on the Windows system), an HTML page titled *VTune(TM) Performance Analyzer Installation CD* shows up. Follow the following steps:

- Select *"Install Now"* for the first item *VTune Performance Analyzer 7.1 Beta*. It'll start the installation process and requires a reboot. This installs VTune for Windows which can be used both for analyzing Windows based applications or controlling other Windows or Linux systems.

- Go to the third item called *VTune Performance Analyzer 7.1 Beta Remote Agents for Linux*. Follow instructions provided in *Installation Instructions* link (to copy the Remote Agent *tar* file to the target Linux system, unpack it and start installation by running *./install.sh* script). The installation on Linux system is very similar to the *Standalone case* described in previous section. So be careful about DCOM, library path and users being added to the *vtune* group.

For starting a performance analysis do this:

- Make sure you have your target application prepared on the Linux system.

- Start the *Linux Remote Agent* by running *[VTunePath]/shared/bin/vtserver*. You should see something like the output in Appendix 1. Remote agent is now ready for accepting connections.

- Launch VTune on the Windows system. Select *New Project*, then *Sampling Wizard* or *Call Graph Wizard*. On the next page, leave *Windows*/Windows*CE/Linux profiling* on and go to the next page.

- You're in step 1 of 3 of the wizard now. Click on the *Remote* button. Type your target IP address in *Machine Name* box. Note if your machine is behind a firewall, enter the gateway IP address here and map port 50000 of your gateway to your target. You might need to do other system admin configuration to expose your target system to the Windows system based on your network configuration. The default port 50000 used by VTune can be overriden. See [7] for more info.

- Select *Linux** from *OS Type* box. Then select *IA-32 architecture* from *Architecture Type* box and press *OK*.

- You're back to step 1 of 3. Now fill in the *Application* box with information about your target application on the Linux system. Note that all path information should be entered as you use it directly from your Linux system like */usr/local/mpich-1.2.5/ch_p4intel/bin/mpirun* for *Application to launch* box, *-np 4 -machinefile my_machines4 cg.A.4* for *Command line arguments* and */home/elec873g2/NPB2.4/NPB2.4-MPI/bin* for *Working directory*. Go to the next page.

- You're in step 2 of 3 of the wizard now. Your target application should show up in *Modules of Interest* box; */usr/local/mpich-1.2.5/ch_p4intel/bin/mpirun* here in our example. Go to the next page.

- You're in step 3 of 3 of the wizard now. Configure it based on your need; for example turn off *Stop collection* condition of 20 seconds or increase it. Press *Finish*.

- Now the VTune on Windows system will contact *vtserver* on the Linux system to start data collection. You should be able to see the incoming connection request on *Linux Remote Agent* and if there is any error generated at application startup. Note that the *Remote Agent* will execute multiple runs of the target application to calibrate its sampling configuration depending on the number of events being captured. When data collection is completed, VTune on Windows system will show histogram of captured events.

For more information about setting up remote connection see *Frequently Asked Questions About Sampling on Linux** in [7]. Also for getting more familiar with VTune concepts and data collectors, See *Getting Started Tutorials* from *Help* menu in VTune Windows.

The properties of the *data collection project* can be changed by right clicking *Activity ...* from *Tuning browser* window. This can be either correction or addition to the list

of modules under profile (and its command line parameters) or change of data collector(s) properties. For example click on *Configure...* button beside *Data Collectors* box. Then add/change the monitored events through *Events* and *Event Ratios* tabs.

6 Test results

Though our goal was not to analyze result of performance data collected through VTune, we here provide the result of some limited test cases without trying to analyze them.

The test system was a Dell PowerEdge 2650s with dual Intel Xeon 2.0 GHz Hyperthreaded Processors with 533MHz front-side bus. CG and BT from NAS Parallel Benchmark (NPB) 2.4 were selected. CG was tried with two cases of 2 and 4 process and BT tested with a 4 process case all from class A.

Surprisingly we noticed that *vtserver* reports only 2 processors instead of expected 4 processors (because of hyperthreaded processors). Apparently hyperthreading was disabled in the system BIOS.

Figures 4 and 5 In Appendix 2 show result for CG class A running with 2 and 4 processes on *a single system* (to be able to monitor it).

The monitored events were:

- Instructions Retired
- μops Retired
- 2nd Level Cache Load Misses Retired
- Misspredicted Branches Retired
- Clockticks
- Streaming SIMD Extensions Input Assist

Inside each picture we copy/pasted the same histogram but with performance data broken for each processor (different color in each bar shows share of a processor). An interesting point is that no SSE instructions detected when running CG with 4 process but a few (5) were detected in 2 process case.

Figure 6 in Appendix 2 shows the same performance events monitored for BT class A running with 4 processes.

We figured out that *Thread view* doesn't work in monitoring Linux applications while the same feature works for Windows-based application (thread view associates collected event samples to individual threads inside a process). Apparently this feature is not supported for Linux (we haven't received a definitive answer from Intel about this yet). This makes VTune not very favorable for detailed performance measurement of OpenMP applications (one of our goals) at this point. But it is expected to have this feature for future releases of VTune.

7 Conclusion

Though *VTune Performance Analyzer* doesn't provide thread-based view for monitoring of hardware performance events under Linux, it has a quite rich environment for performance monitoring of Intel-based processors. All hardware performance events are exposed by VTune and their event mask and threshold-level can be easily programmed. Virtually all low-level performance monitoring register programming can be done through VTune. Controlling the Linux remote agent by a Windows system makes configuration of a performance profile much easier through VTune's user-friendly GUI.

On the other hand, the Linux-based variation of VTune is lagging behind its Windows counterpart in some areas; for example not supporting Thread View, Counter Monitor collector, no GUI and less degree of extensibility. It apparently doesn't hook to system scheduler to provide task-based local-view of performance counters (its weakness compared to PerfCtr). But after all, its ease of use makes it very attractive compared to other performance measurement tools under Linux. We didn't have the chance to explore much with in-depth analysis of an application to trace its behavior and potential bottlenecks through hardware performance counters. This should be the next step in following this project.

References

[1] B. Sprunt, *The basics of performance-monitoring hardware*, IEEE Micro, Volume: 22 Issue: 4 , July-Aug. 2002 Page(s): 64-71.

[2] B. Sprunt, *Pentium 4 performance-monitoring features*, IEEE Micro, Volume: 22 Issue: 4 , July-Aug. 2002 Page(s): 72-82.

[3] Intel, *IA-32 Intel Architecture Software Developers Manual Volume 3: System Programming Guide*.

[4] *Mikael Pettersson's perfctr, a Linux x86 Performance-Monitoring Counters Driver*, http://user.it.uu.se/ mikpe/linux/perfctr/.

[5] *Intel VTune Performance Analyzers Website*, http://www.intel.com/software/products/vtune/.

[6] *PAPI: Performance Application Programming Interface*, http://icl.cs.utk.edu/papi/.

[7] Intel, *Help for Intel VTune Performance Analyzer*, On VTune 7.1 Beta for Windows and VTune 2.0 Beta for Linux.

8 Appendix 1: *vtserver* output after successful start on Linux system waiting for connection from Windows system

```
Setting up ISM environment ...
ISM_DATA_DIR=/tmp/VTune/ISM
ISM_INST_DIR=/opt/intel/vtune/shared/bin
ISM_TEMP_DIR=/tmp/ISM_tmp
Setting up Callgraph environment ...
__Bistro_Exit_Signal__=12
__Bistro_MASTER_PROJECT_PATH__=/home/elec873g2
CLASSPATH =.:/opt/intel/vtune/analyzer/bin
Setting up Sampling environment ...
VTUNE_LINUX_KERNEL_LOAD_ADDRESS=0xC0105000
VTUNE_LINUX_KERNEL_SIZE=0x1F0EC9
VTUNE_LINUX_KERNEL_FILENAME=/boot/vmlinux-2.4.22
Setting up Remote Data Collection environment ...

PATH=/opt/intel/vtune/shared/bin:/opt/intel/vtune/analyzer/bin:/sbin:...
LD_LIBRARY_PATH=/opt/intel/vtune/shared/bin:/opt/intel/vtune/analyzer/bin...
DATA_DIRECTORY=/home/elec873g2
000 12/12/03 10:20:21
===========================================================
000 12/12/03 10:20:21
000 12/12/03 10:20:21 VTune(TM) Performance Analyzer Remote Agent for Linux*
000 12/12/03 10:20:21 Copyright(C) 2003, Intel Corporation, All Rights Reserved
000 12/12/03 10:20:21
000 12/12/03 10:20:21 -- System Information --------
000 12/12/03 10:20:21 number of CPUs:  2
000 12/12/03 10:20:21 CPU speed:  1988 MHz
000 12/12/03 10:20:21
000 12/12/03 10:20:21 -- Remote Agent --------------
000 12/12/03 10:20:21 server version:  v0.9972
000 12/12/03 10:20:21
000 12/12/03 10:20:21 -- Sampling Collector --------
000 12/12/03 10:20:21 driver version:  v0.9193
000 12/12/03 10:20:21 library version:  v0.861
000 12/12/03 10:20:21
000 12/12/03 10:20:21 -- Callgraph Collector -------
000 12/12/03 10:20:21 library version:  v0.91
000 12/12/03 10:20:23
000 12/12/03 10:20:23 -- ISM Agent Proxy -----------
000 12/12/03 10:20:23 library version:  v0.1r
000 12/12/03 10:20:23
000 12/12/03 10:20:23 server on daisy05 (192.168.2.25)
000 12/12/03 10:20:23
000 12/12/03 10:20:23 server listening on port 50000
000 12/12/03 10:20:23
000 12/12/03 10:20:23 server is ready ...
```

9 Appendix 2: Test results

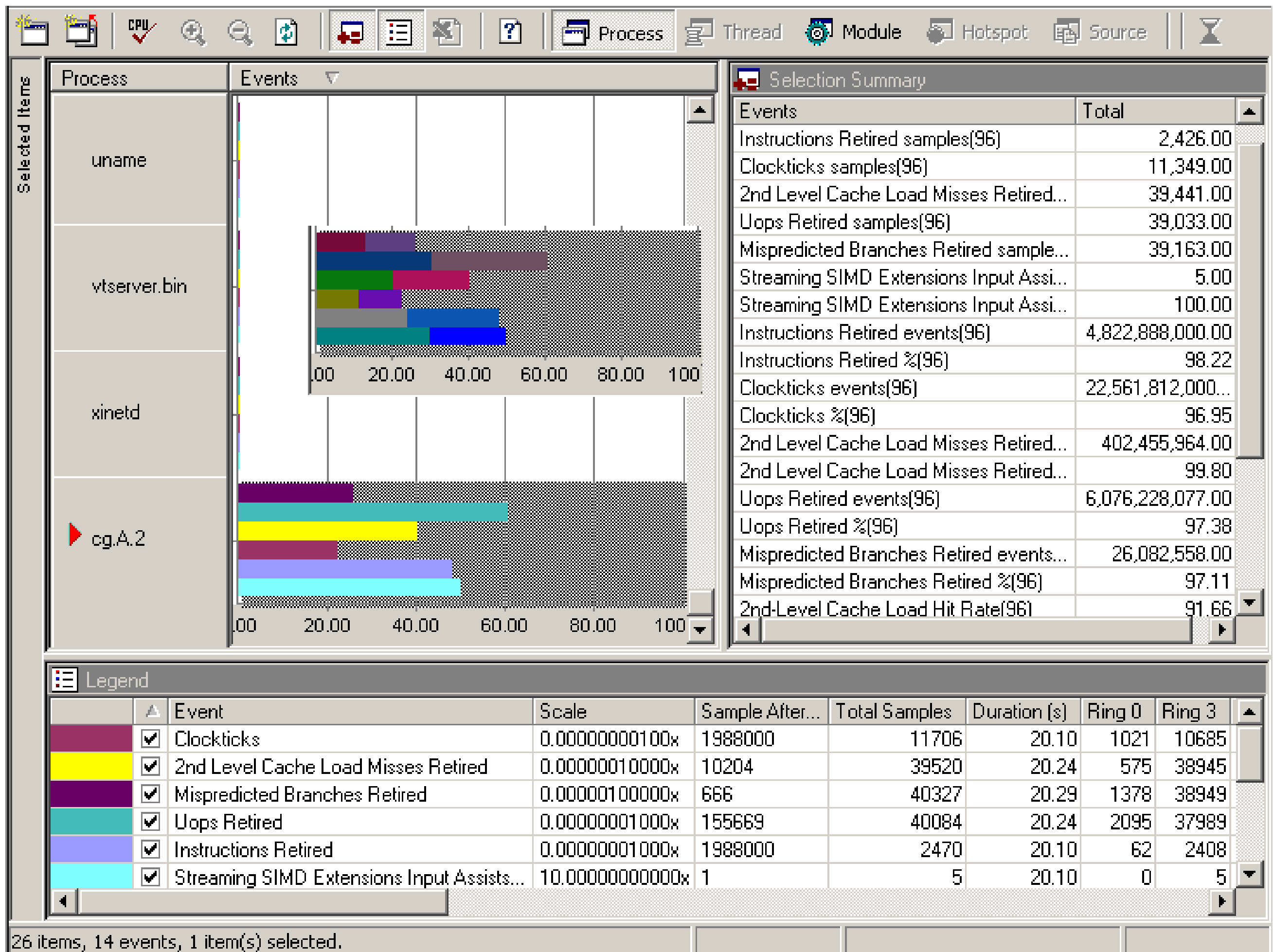

Figure 4: Performance data result for CG class A running on 2 processes

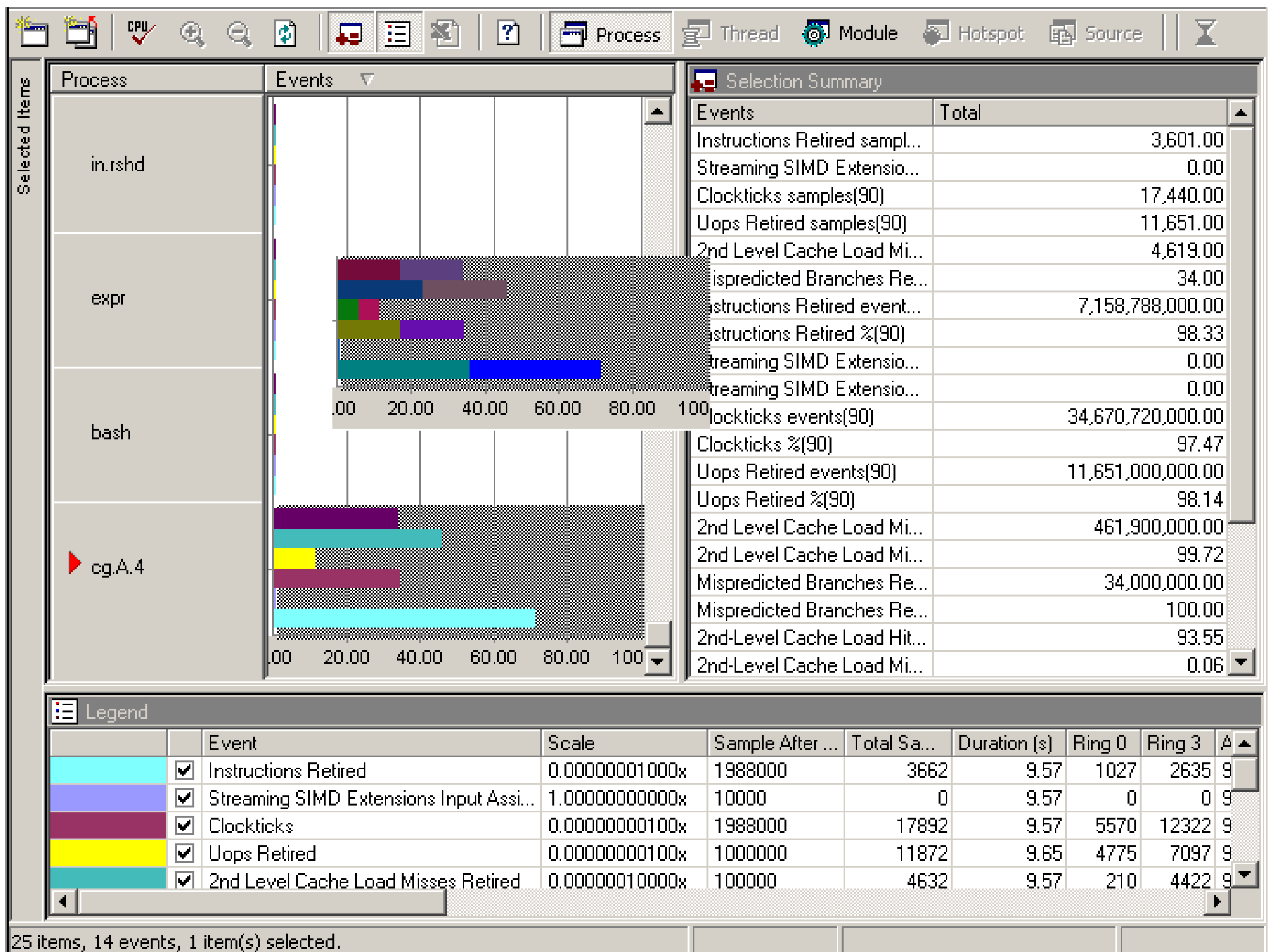

Figure 5: Performance data result for CG class A running on 4 processes

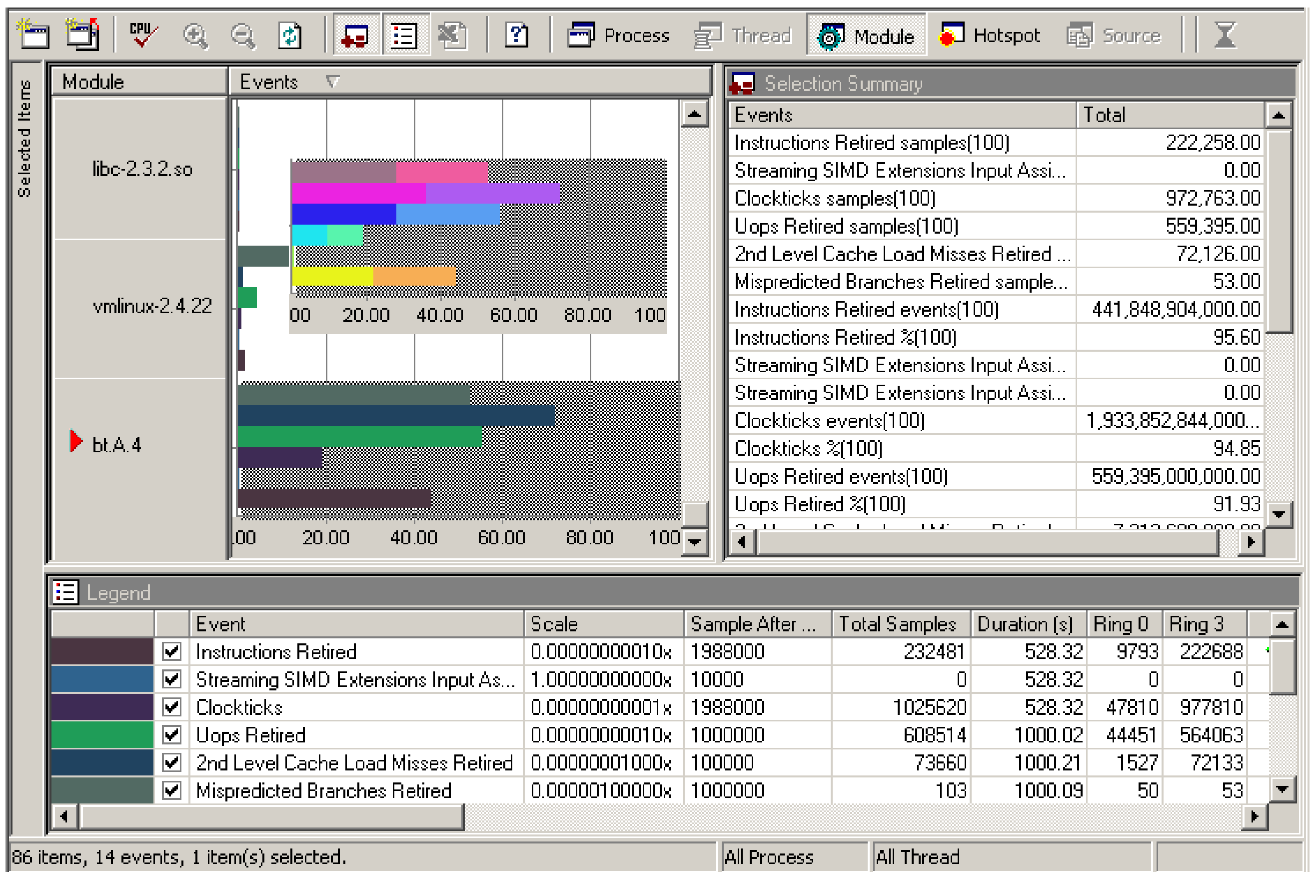

Figure 6: Performance data result for BT class A running on 4 processes