Tutorial: Writing a Linux Trace Toolkit Viewer extension

By Peter Ho

Linux Trace Toolkit Viewer (LTTV) is a free, open source trace visualizer that is used to read, analyze and display traces produced by LTTng. LTTV is extensible via plug-ins. A plug-in is a software tool that accomplishes a specific task in LTTV; for example, the *lttvwindow* plug-in acts as a manager for other plug-ins in LTTV. LTTV consists of several components as shown in Figure 1:

- The main program, defined in the file main.c, is responsible for the creation of the global attributes tree and the parsing of command line options. LTTV has a global attributes tree where hooks are stored; these hooks are used to register callback functions allowing plug-ins to be called at different points during the execution.
- The *lttvwindow* plug-in is a manager that supplies a menu entry and a toolbar for other plug-ins. It also provides library functions upon which graphical plug-ins use to interface with the main window.
- The event viewer displays the detailed information about each event for a given trace.
- The control flow viewer shows the detailed graphical information about each event.
- The statistics viewer displays the statistics for the current traceset.

Figure 1 shows the relationship between the different layers of LTTV.

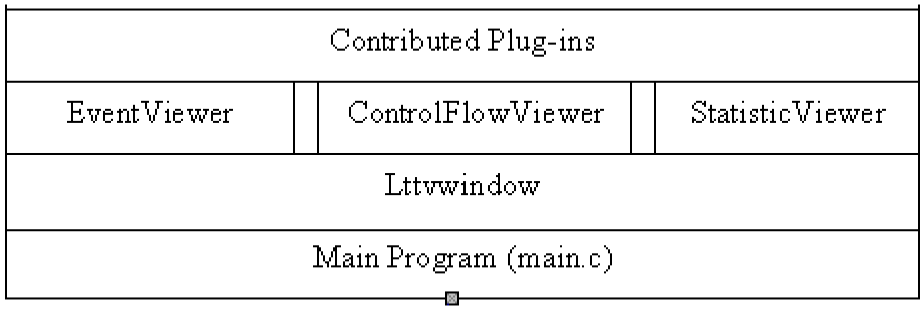


Figure 1: Overview of LTTV

Section 1 and Section 2 provide steps of getting and installing LTTV from the sources. Section 3 explains the process of developing a LTTV graphical plug-in. We start by developing an interrupt plug-in with simple functionalities, the number of interrupts and the total duration. Then we add more complex functionalities such as the standard deviation, the longest and shortest IRQ handlers, the average period, the period and frequency standard deviations. Section 4 shows how to add new instrumentation points with the tool Genevent.

1 Getting Started

In this tutorial, we assume that you have LTTng installed in the kernel. If you haven't done so, see the following resources for some Linux information:

- The Linux Kernel Howto is a good start on how to compile the Linux kernel from the source (http://www.digitalhermit.com/linux/Kernel-Build-HOWTO.html)
- LTTV & LTTng Quick start guide: http://ltt.polymtl.ca/svn/ltt/branches/poly/QUICKSTART
- *Learning Red Hat Linux*, Bill McCarty. O'Reilly. Third Edition
- *Linux kernel development* Robert Love. Novell Press. 2005

This tutorial uses the Linux Fedora distribution. However, other distributions will work as well.

1.1 Getting LTTV

If you have already installed LTTV, you can skip Section 1 and Section 2. If you don't, The LTTV framework is available for download at *http://ltt.polymtl.ca/packages/*. You need to download the LTTV package that is compatible with the installed LTTng. To see the list of compatibilities between LTTV and LTTng, please refer to *http://ltt.polymtl.ca > LTTng+LTTV versions compatibility*.

In the LTTV package, you will find the source code for this tutorial. The simple interrupt plug-in is under *lttv/modules/gui/tutorial*, the interrupt plug-in is under *lttv/modules/gui/interrupt,* and the disk plug-in is under *lttv/modules/gui/diskperformance.*

To configure and compile LTTV, you must obtain the correct compiler, tools and libraries:

- gcc 3.2 or better
- glib 2.4 or better development libraries
- gtk 2.4 or better development libraries

1.1.1 The GLib

GLib or "GNU Library" provides functions such as linked lists, strings, threads, or memory allocation to facilitate the work of the programmer. GLib also provides portability wrappers to allow an application to be more portable on different platforms, such as 16 bits, 32 bits, and big endian or little endian. GLib is not a graphical library and does not require the GTK+ library. You can use it pretty much in any C language program.

To verify which version of GLib is currently installed on your machine:

```
# glibconfig --version
2.4.1
```

This means version 2.4.1 is installed on the machine. If you don't have GLib 2.4 orbetter development libraries, you need to download it from http://rpmfind.net/. The latest version of GLib for program development at time of writing is package *glibdevel2.6.12.i386.rpm*

To install this package, use command:

```
# rpm -iv glibdevel2.6.12.i386.rpm
```

1.1.2 The GTK+

GTK+ is a toolkit designed for the development of applications for the X window System. GTK+ includes GDK which is the acronym for Gimp Drawing Kit. GDK contains functions that encapsulate the functions of Xlib, which is the low-level library of the X window protocol. The GTK+ toolkit proposes a set of widgets that you can use to create graphical user interfaces. In this tutorial, we will use the GTK+ toolkit to create a graphical plug-in for the LTTV framework.

To verify which version of GTK+ is currently installed on your machine:

```
# gtk-config --version
```

If you don't have version 2.4 or higher, you need to download a free copy of GTK+ from http://rpmfind.net. The latest version of GTK+ in rpm format is *gtk2devel2.4.139.i386.rpm*. To install this package, use command:

```
# rpm -iv gtk2devel2.4.139.9.i386.rpm
```

2 Installing and Running LTTV

Now that you have the GLib library and the GTK+ toolkit in place, you can install a LTTV package taken from http://ltt.polymtl.ca/. Unzip the file with the command:

```
# tar xvfz LinuxTraceToolkitViewer-0.x.xx-xxxx2006.tar.gz
# mv  LinuxTraceToolkitViewer-0.x.xx-xxxx2006  lttv
```

Go to the *lttv* directory and use the following commands to compile and install LTTV on your machine:

```
# ./configure
# make
```

Now you need to become root and use the command:

```
# make install
```

The LTTV is now installed on your system! Note that LTTV is built with the GNU autotools package. Under UNIX platforms, the GNU autotools package is used to build portable C and C++ applications. The GNU autotools package is composed of *autoconf*, *automake* and *libtools* programs. The *autoconf* program permits automatic configuration of software installation, it probes the system for portability related information, which is required to customize makefiles, configuration header files, and other application specific files. The output of autoconf is a "configure" shell script. The *automake* program examines source files, determines how they depend on each other, and generates a Makefile so the files can be compiled in the correct order.

If you want to add a new plug-in to LTTV, you need to add the name of the plug-in directory in the Makefile.am file of the parent directory and in the *configure.in* file. The GNU autotools will automatically generate the make files for the new plug-in when you run the *autogen.sh* script.

3 LTTV Plug-ins:

Up until now, we have explained the default LTTV configuration. Now we will write a plug-in to augment the LTTV functionality. There are two types of plug-ins: text and graphical. A text plug-in generates text on the standard output or in a text file, a graphical plug-in is loaded when LTTV starts and contributes to the lttvwindow graphical user interface. In this tutorial, we will concentrate on developing a graphical plug-in.

3.1 Text Plug-in:

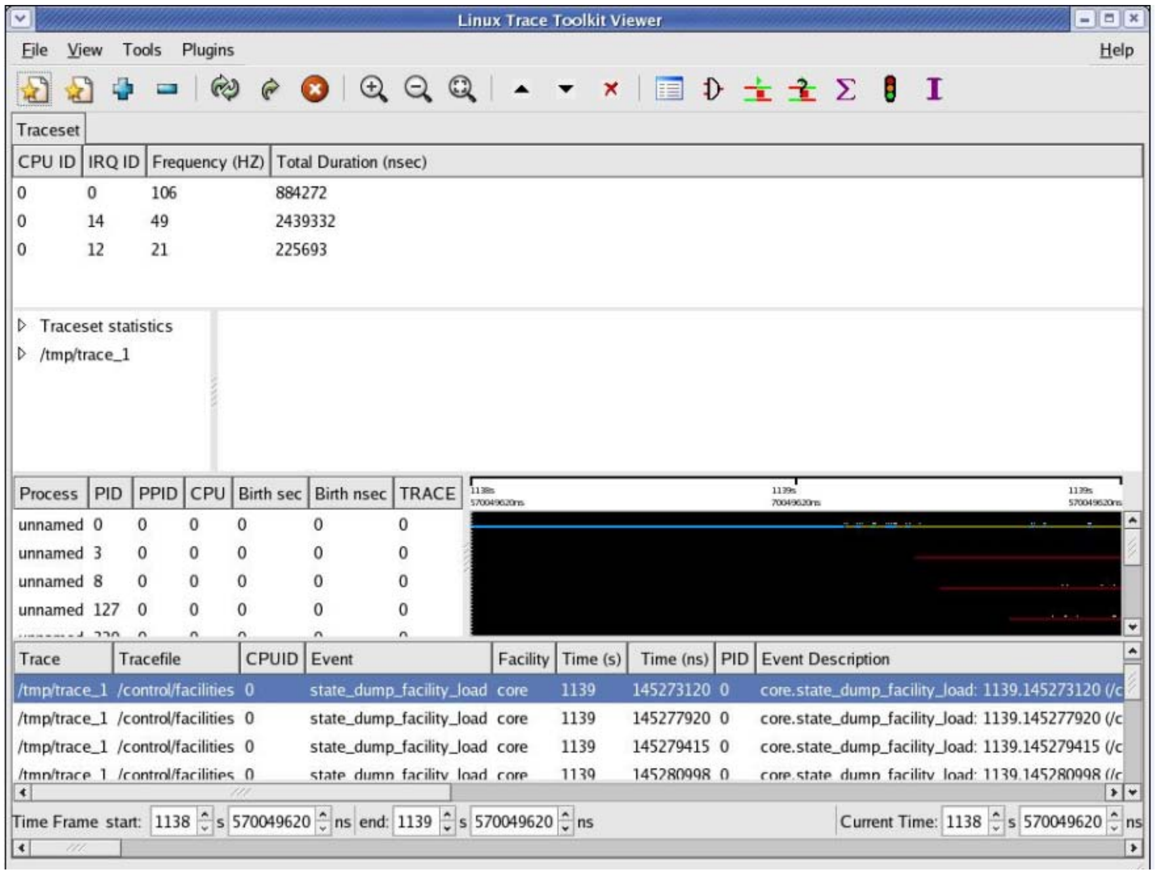
LTTV text plug-ins are located in *lttv/modules/text* directory. The *textdump* plug-in comes with LTTV, it converts the input trace into a formatted text file, and it can be invoked with other plug-ins such as *batchAnalysis* or *textFilter*. Refer to the "*Linux Trace Toolkit Viewer Developer Guide*" on how to develop a text plug-in.

3.2 Simple Graphical Plug-in:

Let's develop an interrupt plug-in to display statistics related to interrupts on the system. An interrupt is an electrical signal that I/O devices use to communicate with the CPU. For example, when you type a key on the keyboard, the keyboard controller will raise an interrupt to signal that a key has been pressed. Watching the frequency and the total duration for each interrupt can give us a rough idea of how a system is performing.

We start by developing a simple version of the interrupt plug-in. This first version will have the frequency of interrupts in Hz and the total duration for each interrupt. The user interface of this version is shown in Figure 2. The first column is the CPU ID, to identify the processor. For example, for a two-processor system, the CPU ID will be 0 or 1. The second column is the IRQ ID; with the i386 PC architecture, there are 16 possible IRQs . Table 1 shows some common IRQs. The third column is the frequency in Hz which is the number of interrupts occurs per second. The fourth column is the total duration in nanosecond which is the sum of all the interrupt time intervals.

In Section 3.3, we will enhance the plug-in by adding the duration standard deviation, the longest and shortest interrupt handlers, the average period, the period and frequency standard deviations.



CPU ID	IRQ ID	Frequency (HZ)	Total Duration (nsec)
0	0	106	884272
0	14	49	2439332
0	12	21	225693

Process	PID	PPID	CPU	Birth sec	Birth nsec	TRACE
unnamed	0	0	0	0	0	0
unnamed	3	0	0	0	0	0
unnamed	8	0	0	0	0	0
unnamed	127	0	0	0	0	0

Trace	Tracefile	CPUID	Event	Facility	Time (s)	Time (ns)	PID	Event Description
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145273120	0	core.state_dump_facility_load: 1139.145273120 (/c
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145277920	0	core.state_dump_facility_load: 1139.145277920 (/c
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145279415	0	core.state_dump_facility_load: 1139.145279415 (/c
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145280998	0	core.state_dump_facility_load: 1139.145280998 (/c

Time Frame start: 1138 s 570049620 ns end: 1139 s 570049620 ns Current Time: 1138 s 570049620 ns

Figure 2: LTTV with the interrupt plug-in located below the toolbar.

IRQ	Purpose
0	Timer
1	Keyboard
2	cascade to IRQ 9
3	COM2 and COM4
4	COM1 and COM3
5	Free
6	Floppy
7	LPT1
8	real-time clock
9	cascade from IRQ2
14	Hard Drive

Table 1: PC interrupt

The source code of the plug-in is available in the directory *lttv/modules/gui/tutorial* of the LTTV main branch. You might want to download the file *tutorial.c* and have a look at it before we start.

Inner Data Structure:

First, we need a data structure to hold the plug-in information. Usually, this data structure must contain fields for the following information: the GTK widgets for the graphical interface, the event hooks registered with the main program, and internal data specific to the information to display. This structure will have to be passed as hook_data to each function registered by the plug-in.

The data structure of our plug-in is along the lines of the following code snippet:

```
typedef struct _InterruptEventData {

/*Graphical Widgets */
GtkWidget * ScrollWindow;
GtkListStore *ListStore;
GtkWidget *Hbox;
GtkWidget *TreeView;
GtkTreeSelection *SelectionTree;

Tab             * tab; /* tab that contains this plug-in*/
LttvHooks  * event_hooks;
LttvHooks  * hooks_trace_after;
LttvHooks  * hooks_trace_before;
TimeWindow   time_window;
LttvHooksById * event_by_id_hooks;
GArray *IrqExit;
  GArray *IrqEntry;
} InterruptEventData ;
```

The graphical user interface should contain a table within a scrolled window. It starts with the top-level window, where we create a ScrollWindow. The ScrollWindow is a place within which other widgets will be placed. This top-level window is actually a container for the Hbox and the Treeview. The Hbox is a container that organizes child widgets into a single row, we want the child widgets to horizontally align. A Treeview widget can display both trees and tabular lists, in our case, we use it to display lists. A

ListStore object is a list model for use with a GtkTreeView widget. Figure 3 illustrates this.

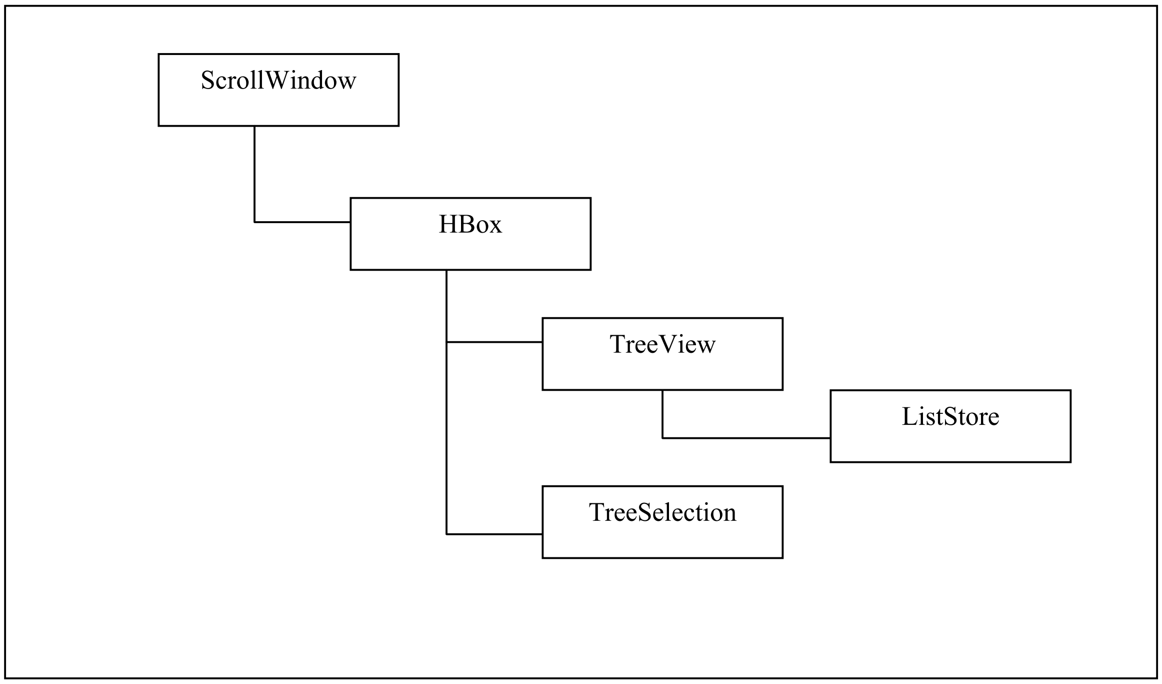


Figure 3: Widget Instance Hierarchy

Figure 3 illustrates the widget hierarchy for our user interface. The Scrollwindow at the top of the hierarchy is the parent to widgets residing lower in the hierarchy.

Entry point *init()*

Basically, the entry point of a LTTV plug-in is the *init()* function, which is called when the plug-in is initialized. Then it calls *lttvwindow_register_constructor()*:

```
void lttvwindow_register_constructor(
        char * name,
        char * menu_path,
        char * menu_text,
        char ** pixmap,
        char * tooltip,
        lttvwindow_viewer_constructor view_constructor );
```

This function is defined in *lttwindow.c* and is responsible to register the plug-in's constructor, add the pixmap file to the toolbar of the main window and create a menu item for the extension on the menu bar of the main window. The *name* argument is the name of the plug-in. The *menu_text* argument is the text of the menu item. The *pixmap* is the .xpm pixmap on the toolbar item. The *view_constructor* is a pointer to the callback function that you want called. This provides everything needed to make the new viewer provided by the extension available.

Our callback function *InterruptEventData *system_info(Tab *tab)* is responsible for the allocation and initialization of the data structure *InterruptEventData*, initialization of GTK widgets, and register hook functions to event hooks. It is called when an Interrupt plug-in is requested from the menu. It also calls *lttvwindow_register_time_window_notify()* to register a hook function that will be called by the main window when the main window's time interval is updated. The function is defined in lttvwindow.c as follows:

```
void lttvwindow_register_time_window_notify(Tab *tab,
                                    LttvHook    hook,
                                    gpointer    hook_data);
```

The *tab* argument is a pointer to the tab folder where the extension belongs. The *hook* argument points to a hook function. In our case, we register the *interrupt_update_time_window()* function, so it will be called by the main window whenever the user changes the time interval with the time scrollbar at the bottom of the LTTV window. This synchronizes the time intervals for all the viewers within the same window.

Request events

A plug-in can request events by passing an EventsRequest structure to the main window. To do this, the plug-in must pass an initialized EventsRequest structure to the function *lttvwindow_events_request()*. This function is responsible to request data in a specific time interval to the main window and is defined in *lttwindow.c* as follows:

```
void lttvwindow_events_request(Tab *tab,
                                      EventsRequest  *events_request);
```

The *tab* argument is the tab folder where the plug-in belongs to. The *events_request* argument is the pointer to an EventsRequest structure . This structure must be initialized with the start time or the start position, the end time or the end position, and the hooks associated with the event request.
Here is the EventsRequest structure:

```
typedef struct _EventsRequest
{
  gpointer                owner;            /* Owner of the request       */
  gpointer                viewer_data;      /* Unset : NULL               */
  gboolean                servicing;        /* service in progress: TRUE*/
  LttTime                 start_time;       /* Unset : ltt_time_infinite   */
  LttvTracesetContextPosition *start_position;  /* Unset : NULL           */
  gboolean                stop_flag;              /* Continue:TRUE Stop:FALSE */
  LttTime                 end_time;               /* Unset : ltt_time_infinite   */
  guint                   num_events;             /* Unset : G_MAXUINT       */
  LttvTracesetContextPosition *end_position;  /* Unset : NULL             */
  gint                    trace;                  /* unset : -1            */
  GArray                  *hooks;                 /* Unset : NULL           */
  LttvHooks               *before_chunk_traceset; /* Unset : NULL       */
  LttvHooks               *before_chunk_trace;     /* Unset : NULL       */
  LttvHooks               *before_chunk_tracefile;/* Unset : NULL       */
  LttvHooks               *event;                 /* Unset : NULL           */
  LttvHooksById           *event_by_id;      /* Unset : NULL           */
  LttvHooks               *after_chunk_tracefile; /* Unset : NULL       */
  LttvHooks               *after_chunk_trace;     /* Unset : NULL       */
  LttvHooks               *after_chunk_traceset;  /* Unset : NULL       */
  LttvHooks               *before_request;  /* Unset : NULL           */
  LttvHooks               *after_request;   /* Unset : NULL           */
} EventsRequest;
```

Some relevant fields are:
The *owner* field is a pointer to the extension's data structure; in our case, it's the InterruptEventData structure. The *start_time* field is a pointer the start time interval of the main window. The *end_time* field is a pointer the end time interval of the main window
The *before_chunk_traceset, before_chunk_trace, before_chunk_tracefile, event, after_chunk_tracefile, after_chunk_trace and after_chunk_traceset* fields are hook pointers for traceset, trace and tracefile. In LTTV, a traceset is a set of traces; this means you can have many traces in one traceset. One trace can contain many tracefiles. The *before* means that the hooks will be called before the beginning of a trace, traceset or tracefile and the *after* is the opposite. The *event* field is a hook pointer for an event in a trace. The *event_by_id* field is a hook pointer for the event ID.

With this in mind, we look at the *request_event ()* function of our plug-in. This function gets a traceset from the traceset context. Since there are many traces in a traceset, we need to iterate through the traceset. For each trace, we create a hook for each hook function, we use two hook functions: *trace_header*(), and *interrupt_display()*. The *trace_header()* function is called at the beginning of the trace but it does nothing, the *interrupt_display ()* function displays the result on the viewer.

Use event ID

To calculate the number of interrupts and the total duration for each interrupt, we need catch the events "irq_entry" and "irq_exit" from the trace. There are two ways to do this. The easy way is to register a hook function to an event hook, and assigns the hook to the *event* field of *EventsRequest* structure. The pseudo code should look like this:

```
static void request_event(InterruptEventData *event_data )
{
  .
  .
  .
  // Create an event_hooks
  event_data->event_hooks = lttv_hooks_new();
  //  Register  parse_event() callback function to the event_hooks
  lttv_hooks_add(event_data->event_hooks, parse_event, event_data, LTTV_PRIO_DEFAULT);
  .
  .
  EventsRequest *events_request = g_new(EventsRequest, 1);
  events_request->event              = event_data->event_hooks;
  .
  .
  lttvwindow_events_request(event_data->tab, events_request);

}

gboolean parse_event(void *hook_data, void *call_data)
{
   extract an event from call_data
   if the event is an "irq_entry" or "irq_exit" event
   then calculate the duration of each interrupt
}
```

Whenever an event occurs in the trace, the function *parse_event ()* is called. Then it filters for the "irq_entry" and "irq_exit" events, which are used to calculate the duration of each interrupt and to count the interrupt.

The more elegant way is to use the event ID. With the event ID approach, the registered hook function is called only whenever the registered event occurs. We use this method for the implementation of our interrupt plug-in. The following pseudo code of our *request_event()* illustrates the event ID concept:

```
static void request_event(InterruptEventData *event_data )
{
  .
  .
  .
 //Create an array of LttvTraceHook
  GArray *hooks  = g_array_new(FALSE, FALSE, sizeof(LttvTraceHook));

  EventsRequest * events_request = g_new(EventsRequest, 1);

  hooks = g_array_set_size(hooks, 2)

  LttvTraceState *ts = = (LttvTraceState *)tsc->traces[i];

  event_data->event_by_id_hooks = lttv_hooks_by_id_new();

  lttv_trace_find_hook(ts->parent.t,
                          LTT_FACILITY_KERNEL, LTT_EVENT_IRQ_ENTRY,
                          LTT_FIELD_IRQ_ID, 0, 0,
                          irq_entry_callback,
                          events_request,
                          &g_array_index(hooks, LttvTraceHook, 0));

 lttv_trace_find_hook(ts->parent.t,
                          LTT_FACILITY_KERNEL, LTT_EVENT_IRQ_EXIT,
                          LTT_FIELD_IRQ_ID, 0, 0,
                          irq_exit_callback,
                          events_request,
                          &g_array_index(hooks, LttvTraceHook, 1));

 // Iterate through the facility list
  for(k = 0 ; k < hooks->len; k++)
  {
      hook = &g_array_index(hooks, LttvTraceHook, k);
      for(l=0; l<hook->fac_list->len; l++)
      {
          thf = g_array_index(hook->fac_list, LttvTraceHookByFacility*, l);
          lttv_hooks_add(lttv_hooks_by_id_find(event_data->event_by_id_hooks, thf->id),
                                      thf->h,
                                      event_data,
                                      LTTV_PRIO_DEFAULT);

      }
  }
  .
  .
  .
  events_request->hooks = hooks;
  events_request->event_by_id = event_data->event_by_id_hooks;
  .
  .
  .
  lttvwindow_events_request(event_data->tab, events_request);
}
static gboolean irq_entry_callback(void *hook_data, void *call_data)
{
}
static gboolean irq_exit_callback(void *hook_data, void *call_data)
{

}
```

We start by creating an array of *LttvTraceHook* with the size of 2 for the "irq_entry" and "irq_exit" events. Each item of this array is used to store information returned by the *lttv_trace_find_hook*() function, which is defined as follows:

```
gint lttv_trace_find_hook(LttTrace *t,
                           GQuark facility,
                          GQuark event_type,
                           GQuark field1, GQuark field2, GQuark field3,
                          LttvHook h,
                          gpointer hook_data,
                          LttvTraceHook *th);
```

This function searches in the trace for the id of the named event type within the named facility. Then, find the three (if non null) named fields. All that information is then used to fill the LttvTraceHook structure.

The default facilities and events quarks, such as LTT_FACILITY_KERNEL, LTT_EVENT_IRQ_EXIT, and LTT_EVENT_IRQ_ENTRY , are defined and initialized in *state.c*. If you are to create new facilities or events, you define them directly in your plug-in as was done in the disk plug-in (*diskperformance.c*).

For each item of the *LttvTraceHook* array, we need to iterate through its facility list. This is because we may have more than one facility associated with a facility name in a trace. For example, LTTV registers the facility "myfacility" with an ID 10, then unregisters the same facility “myfacility” a moment later. The ID 10 will be unused for the rest of the trace. Then it registers the facility “myfacility” again with a new ID 11. When you register your event for “myfacility”, you must connect the event ID to the facility IDs 10 and 11. This is why we iterate through the facility array and call *lttv_hooks_by_id_find( )* to obtain the hooks for a given ID.

```
events_request->hooks = hooks;
```

Here, we need to set the LttvTraceHook array to the *hooks* field of the EventsRequest structure. This allows the main window to free the memory allocated for the hooks. Otherwise, we will create a memory leak.

3.3 Improved Plug-in

We can enhance the interrupt plug-in by adding the duration standard deviation, the longest and shortest interrupt handlers, the average period, the period and frequency standard deviations.

The source code of the interrupt plug-in is available in directory *lttv/modules/gui/interrupt* of the LTTV main branch. You might want to download the *interrupt.c* file and have a look at it before we start.

To make the explanation simple, we will concentrate only on the duration standard deviation calculation. The standard deviation measures the average distance of the data values from their mean (average). If the data points are all close to the mean, then the standard deviation will be low. We can use this to determine whether an interrupt is periodic or not, if its standard deviation is low or zero, then we can conclude that the interrupt is periodic.

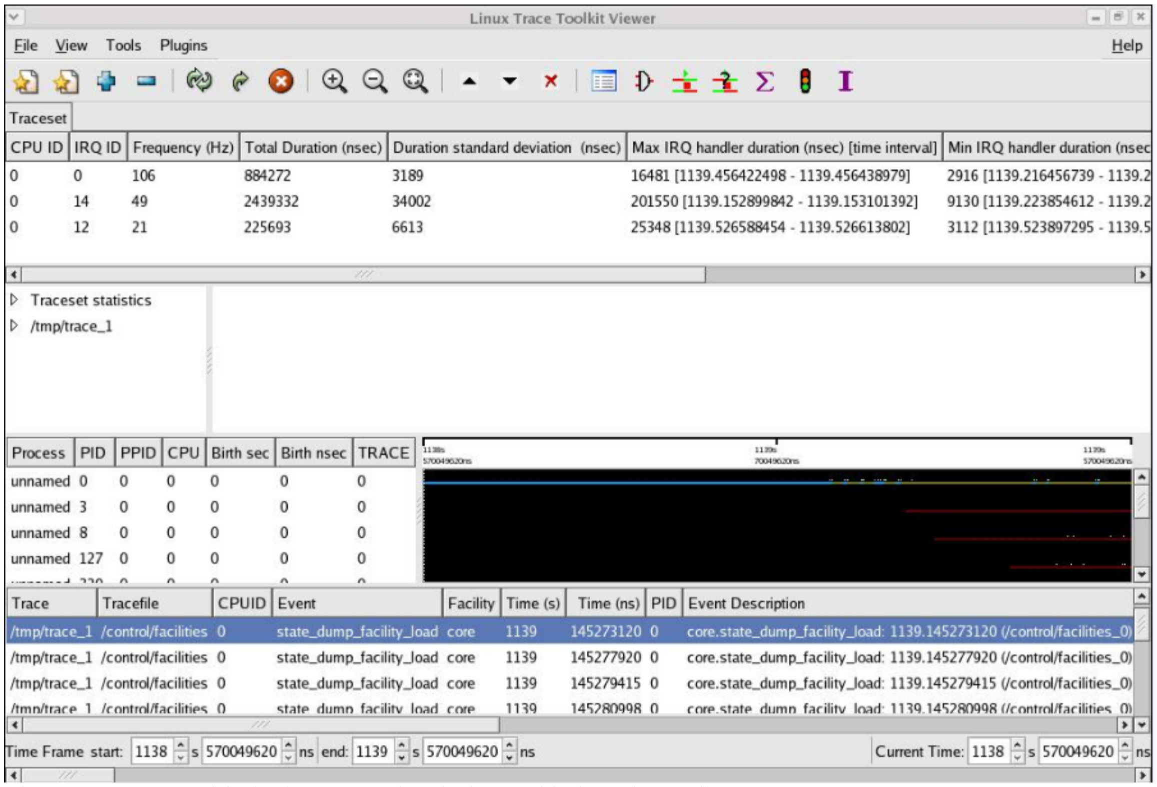
The standard deviation is defined as:

$$s = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

where $\overline{x}$ is the average duration and x_i is the individual interrupt duration. This formula is taken from the Wikipedia encyclopedia.

To calculate the standard deviation, we need to iterate through the event trace twice. In the first iteration, we need to calculate the average $\overline{x}$. In the second iteration, we use the average $\overline{x}$ to compute the standard deviation *S*. To translate this into programming code, we need to pass the EventsRequest structure twice to the main window. In the first request, we setup a hook function to count the number of interrupts, which we use to calculate the average $\overline{x}$. In the second request, which is called immediately after the first request is done, we have a function hook to calculate standard deviation *S*. If you look at the file *interrupt.c*, I call the two requests: *FirstRequest( )* and *SecondRequest()*, respectively. For both requests, I use the event ID concept explained previously.

The result of the new plug-in is shown in Figure 4. The fifth column is the duration standard deviation. The sixth and seventh columns represent maximum and minimum IRQ handler durations. The eighth column is the average period.



CPU ID	IRQ ID	Frequency (Hz)	Total Duration (nsec)	Duration standard deviation (nsec)	Max IRQ handler duration (nsec) [time interval]	Min IRQ handler duration (nsec
0	0	106	884272	3189	16481 [1139.456422498 - 1139.456438979]	2916 [1139.216456739 - 1139.2
0	14	49	2439332	34002	201550 [1139.152899842 - 1139.153101392]	9130 [1139.223854612 - 1139.2
0	12	21	225693	6613	25348 [1139.526588454 - 1139.526613802]	3112 [1139.523897295 - 1139.5

Process	PID	PPID	CPU	Birth sec	Birth nsec	TRACE
unnamed	0	0	0	0	0	0
unnamed	3	0	0	0	0	0
unnamed	8	0	0	0	0	0
unnamed	127	0	0	0	0	0

Trace	Tracefile	CPUID	Event	Facility	Time (s)	Time (ns)	PID	Event Description
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145273120	0	core.state_dump_facility_load: 1139.145273120 (/control/facilities_0)
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145277920	0	core.state_dump_facility_load: 1139.145277920 (/control/facilities_0)
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145279415	0	core.state_dump_facility_load: 1139.145279415 (/control/facilities_0)
/tmp/trace_1	/control/facilities	0	state_dump_facility_load	core	1139	145280998	0	core.state_dump_facility_load: 1139.145280998 (/control/facilities_0)

Time Frame start: 1138 s 570049620 ns end: 1139 s 570049620 ns Current Time: 1138 s 570049620 ns

Figure 4: LTTV with the interrupt plug-in located below the toolbar.

4 Adding new instrumentation events

Sometimes you may need to add new instrumentation events to the instrumented kernel. The LTT Next Generation (LTTng) provides a simple mechanism to do this with a tool called *genevent*. Genevent is a XML parser that takes as input a XML event description file and from this creates C language files, that LTTng uses for instrumentation. Basically, you create a XML event description file, the *genevent* tool will generate appropriate headers from this XML file to use for tracing.

For example, we want to create a XML event description for the block facility contains two events, *read* and *write,* as shown in Figure 5. The *read* event counts the number of bytes read in a block operation and has three fields: *major, minor,* and *bytes.* The *major* and *minor* fields identify the block device, and the *bytes* field identifies the number of bytes transferred. The same reasoning applies to the *write* event.

```
<facility name=block>
 <description>block facility has events related to block read and block written.</description>


  <event name=read>
   <description>block read event</description>
   <field name="major"> <description>major number of the device</description> <int/> </field>
   <field name="minor"> <description>minor number of the device</description> <int/> </field>
   <field name="bytes"> <description>number of bytes read</description> <size_t/> </field>
 </event>
    <event name=write>
   <description>Block write event </description>
   <field name="major"> <description>major number of the device</description> <int/> </field>
   <field name="minor"> <description>minor number of the device</description> <int/> </field>
   <field name="bytes"> <description>number of bytes written</description> <size_t/> </field>
 </event>
 </facility>
```

Figure 5: XML event description file: *block.xml*

I save the *block.xml* file in lttv/modules/gui/diskperformance of the LTTV main branch.

The process of adding new instrumentations is as follows:

- Install *genevent*. Make sure you check the compatibility list for the appropriate version.
- Create a facility XML file
- Generate appropriate source code from the XML file with genevent
- Copy the source code to /usr/src/linux-2.6.x-lttng-x.x.x/include/linux/ltt and /usr/src/linux-2.6.x-lttng-x.x.x/ltt
- Edit the kernel config file */usr/src/linux-2.6.x-lttng-x.x.x/ltt/Kconfig* by adding

your config flag

- Edit the make file */usr/src/linux-2.6.x-lttng-x.x.x/ltt/Makefile* by adding your loader name
- Edit the kernel file you want to instrument:
  - Add #include <linux/ltt/ltt-facility-xxx.h> at the beginning of the file.
  - Add a call to the tracing functions. See their names and parameters in /usr/src/linux-2.6.x-lttng-x.x.x/include/linux/ltt/ltt-facility-xxx.h

Follow this process, copy the *block.xml* file to */usr/local/share/LinuxTraceToolkitViewer/facilities*. Use genevent to generate LTTng files for the kernel:

# cd /tmp

# genevent /usr/local/share/LinuxTraceToolkitViewer/facilities/block.xml

This command generates the following files: *ltt-facility-block.h, ltt-facility-id-block.h, ltt-facility-loader-block.h, ltt-facility-loader-block.c*. Copy the first two files to the *kernel-2.6.x/include/linux/ltt* directory and the last two files to *kernel-2.6.x/ltt* directory of the instrumented kernel.

You need to add the LTT_FACILITY_BLOCK flag in the *kernel-2.6.x/ltt/Kconfig* file and the CONFIG_LTT_FACILITY_BLOCK flag in the *kernel-2.6.x/ltt/Makefile*. This step is necessary for the new facility to be compiled with LTTng in the kernel.

Now, we need to place the *read* and *write* events at the block I/O layer of the kernel to get the number of bytes transferred. Let's look at the *read* operation in the Linux kernel. Whenever a block device is opened, the kernel calls the blk_open() function, defined in fs/block_dev.c, this is where we get the minor and major numbers of the device. All read operations from a block device would have eventually called the *__generic_file_aio_read()* function, defined in mm/filemap.c. In turn, this function calls *do_generic_file_read()* to perform the read operation from the disk and it returns a *read_descriptor_t* structure. The *trace_block_read()* function, generated by *genevent* tool, should be placed after the *do_generic_file_read()* as shown in the following code snippet:

```
ssize_t __generic_file_aio_read(struct kiocb *iocb, const struct iovec *iov, unsigned long nr_segs, loff_t
*ppos)
{
.
.
.
        do_generic_file_read(filp,ppos,&desc,file_read_actor);
        retval += desc.written;
        trace_block_read( _major,_first_minor, retval);
        if (desc.error) {
                retval = retval ?: desc.error;
                break;
        }
}
```

The same reasoning applies for the written operation. Place the *trace_block_write()* function in the function *__generic_file_aio_write_nolock ()* of the file *mm/filemap.c* as shown in the following code snippet:

```
static ssize_t __generic_file_aio_write_nolock(struct kiocb *iocb, const struct iovec *iov,
                                        unsigned long nr_segs, loff_t *ppos)
{
.
.
.
        written = generic_file_buffered_write(iocb, iov, nr_segs, pos, ppos, count,
                                        written);
        trace_block_write(_major, _first_minor, written );
out:
        current->backing_dev_info = NULL;
        return written ? written : err;
}
```

After we recompile the instrumented kernel and get a new trace with LTTV, we should be able to view the new instrumentation events with the disk performance viewer.

The LTTV disk performance plug-in is used to analyze traces containing the *block* facility with the *read* and *written* events. It displays statistics on the IDE, SCSI and USB disk devices. The statistics included the total bytes read and written, the number of read and write operations, and the rate of transferred. Analyzing these statistics can show us uneven load of multiple disks, and show the balance of read versus write operations.

The disk performance plug-in source code is located in the directory *lttv/modules/gui/diskperformance* of the LTTV main branch. The source code of this plug-in is self-explained, as it used the event ID concept explained previously.

Download source code:

Linux Trace Toolkit Viewer (LTTV) packages: http://ltt.polymtl.ca/packages/

LTT Next Generation (LTTng): http://ltt.polymtl.ca/lttng/

Acknowledgments:

I like to thank Mathieu Desnoyers for his technical expertise and reviews.

Bibliography:

[1] Mathieu Desnoyers, “Linux Trace Toolkit Viewer Developer Guide”

[2] Mathieu Desnoyers, “Linux Trace Toolkit Viewer User Guide”

[3] Mathieu Desnoyers, “LTTng & LTTV QuickStart Guide”