Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Review Committee

Proceedings Formatting Team

Driver Tracing Interface

David J. Wilder
IBM Linux Technology Center
dwilder@us.ibm.com

Michael Holzheu
IBM Linux on zSeries Development
holzheu@de.ibm.com

Thomas R. Zanussi
IBM Linux Technology Center
zanussi@us.ibm.com

Abstract

This paper proposes a driver-tracing interface (DTI) that builds on the existing Relay tool and the proven Debug Feature model used by IBM™ zSeries Linux. Users of this infrastructure are provided with individual, manageable channels for capturing or passing debug data into user space. Separate channels are created by each subsystem or driver. Data is stored in kernel ring buffers providing *flight recorder* type functionality. Unwanted or unconsumed data is simply discarded where pertinent data can be saved for future analysis. In the instance of a system crash, all unconsumed tracing data is automatically saved in crash dumps. With support from crash analysis tools like crash or lcrash, trace data can be extracted directly from a crash dump, providing an exact trace of the events leading up to the crash.

Developers of Linux™ device drivers will be interested in DTI as a tool to aid in the troubleshooting of their drivers. Service engineers and support personnel who are tasked with isolating driver bugs will learn how to capture DTI data on a live system and extract DTI data from a crash dump.

1 Introduction

Webster defines a trace as "the track left by the passage of a person, animal, or object." Applied to computer systems, we can adapt this definition to mean the track left by the execution of a program. A typical program doesn't normally leave tracks, other than the expected side effect of the program. To cause a program to create tracks so that its passage can meaningfully be tracked, code that explicitly leaves those tracks must be added into the execution path of the program. We call the individual tracks we've inserted *tracepoints*. Two types of tracepoints can be used.

- **Static:** Tracks that are added to the source code and compiled with it.
- **Dynamic:** Tracks that are added to the execution stream at run time.

Tracing is the act of causing special-purpose code associated with a program to report something specific about what the program is doing at a given point. The information can be simple or complex, high or low-frequency, binary or text-based, time-based or unsequenced and so on. The resulting data stream can be continuously persisted to long-term storage, sent to a destination over a network connection, or it can be endlessly cycled around a constant-sized buffer. The buffer will only be read when an event of interest occurs and a user needs details about the sequence of events that led up to that event, for example a system crash or afailed assertion in the normal program flow.

1.1 Why is tracing needed

Tracing is needed because, in many situations, only a detailed, sequenced, or timestamped history of program execution can explain the behavior of a program or the pathology of a problem. In many cases, coarse-grained statistical or summary information can show the general area of a problem, but only trace data can show the true source of the problem.

The detailed data from a complete trace can be post-processed and summary information or aggregated statistics can be calculated based on it. The converse however is not true. Detailed information cannot be extracted from statistics or summaries, because that information is lost in the process. Keeping in mind practical considerations such as storage costs, it is always better to have the detailed trace information instead of only the statistics, because the appropriate trace data allows for more varied and flexible analysis.

2 Current solutions for static tracing

Currently, there are three kernel APIs available to do static tracing using kernel memory buffers:

- printk
- relay
- s390 debug feature (s390dbf)

2.1 Printk

Printk often is misused for tracing purposes, since there is no other standard way for device drivers to log debug information.[1] There are multiple printk levels defined, which indicate the importance of a kernel message. The kernel messages are written in one global printk buffer, which can be accessed from the user space with the `syslog()` system call or via the */proc/kmesg* file. The dmesg tool prints the content of the message buffer to the screen, and then the kernel log daemon klogd reads kernel messages and redirects them either to the syslogd or into a file. The printk message buffer can also be accessed from system dumps through the lcrash or other crash dump analysis tools.

2.2 Relay

Relay provides a basic low-level interface that has a variety of uses, including tracing. In order to define a kernel trace buffer, the `relay_open()` function is used. This function creates a relay channel. Relay channels are organized as wrap around buffers in memory. There are two mechanisms to write trace data into a channel:

- `relay_write(chan, data, length)` is used to place data in the global buffer.
- `relay_reserve(chan, length)` is used to reserve a slot in a channel buffer which can be written to later.

Relay buffers are represented as files in a host file system such as debugfs; data previously written into a relay channel can be retrieved by read(2)ing or mmap(2)ing these files.

2.3 The s390 debug feature

The s390 debug feature (s390dbf) is a tracing API, which is used by most of the s390 specific device drivers. Each device driver creates its own debug feature in order to log trace records into memory areas, which are organized as wrap around ring buffers. The s390dbf uses its own ring buffer implementation. The main purpose of the debug feature is to inspect the trace logs after a system crash. Dump analysis tools like crash or lcrash can be used to find out the cause of the crash. If the system still runs but only a subcomponent which uses dbf fails, it is possible to look at the debug logs on a live system via the Linux debugfs file system.

Device drivers can register themselves to the debug feature with the `debug_register()` function. This function initializes an s390dbf for the caller. For each s390dbf there are a number of debug areas where exactly one is active at one time. Each debug area consists of a set of several linked pages in memory. In the debug areas, there are stored debug entries (trace records) which are written by event and exception calls.

An event call writes the specified debug entry to the active debug area and updates the log pointer for the active area. If the end of the active debug area is reached, a wrap around in the ring buffer is done and the next debug entry will be written at the beginning of the active debug area.

An exception call writes the specified debug entry to the log and switches to the next debug area. This is done in order to guarantee that the records that describe the origin of the exception are not overwritten when a wrap around for the current area occurs.

The debug areas themselves are also ordered in the form of a ring buffer. When an exception is thrown in the last debug area, the next debug entries are then written again in the very first area.

Each debug entry contains the following data:

- Timestamp

[1] This paper is not purposing a replacement for printk. The DTI is purposed as an additional tool that should be used in place of printk only when true tracing is needed.

- Cpu-Number of calling task
- Level of debug entry (0...6)
- Return Address to caller
- Flag that indicate whether an entry is an exception or not

The trace logs can be inspected in a live system through entries in the debugfs file system. Files in the debugfs that were created by s390dbf represent different views to the debug log. The purpose of s390dbf views is to format the trace records in a human-readable way. Predefined views for hex/ascii, sprintf and raw binary data are provided. It is also possible to define other component specific views. The content of a view can be seen by reading the corresponding debugfs file. The standard views are also available in the dump analysis tools lcrash and crash. Figure 1 shows an example of an s390dbf sprintf view.

3 Driver Tracing Interface

This section describes the proposed Driver Tracing Interface (DTI) for the Linux kernel. It starts by examining the project goals and usage models that were considered in its design. Also provided is a brief description of two existing subsystems that DTI depends on, the DebugFS and relay subsystems.

3.1 Design goals

DTI will add supportability to drivers and subsystems that adopt it. However, from the support community viewpoint the deployment of DTI must be propagated into a significant number of subsystems, drivers, and system architectures before it usefulness is proven. Developing support tools and process around one off solutions is costly and unproductive to support organizations, therefore a key aspect of our project goals is to provide a feature that can easily be adopted by the Linux development community, thus ensuring its wide use.

- The DTI's API should be as simple and easy to implement as possible.
- DTI should be architecture independent.

Adding code into a driver or subsystem that is not contributing to the core functionality might be seen as unnecessary. This concern must be addressed by DTI, ensuring that the added benefit is balanced with low overhead of the feature.

- DTI should have as little performance impact as possible.
- DTI should reuse existing code in the Linux kernel.
- DTI must implement per-CPU buffering.
- DTI should minimize the in-kernel processing of trace data.

The remaining goals specify functionality requirements.

- DTI's API should be usable in both user context and interrupt context.
- Trace data must be buffered so that it can be retrieved from a crash dump.
- Trace data must be viewable from a live system.
- DTI must allow for a rich set of tools to process trace data.

3.2 Usage models

Two primary usage models were examined when designing the DTI API.

Usage Model 1: **Isolating a driver problem from a post-mortem cash dump analysis.** In this scenario, the system has crashed and a crashed system image (crash dump) has been obtained. By analyzing the crash dump the user suspects there is a problem in the XYZ driver. Using the DTI commands integrated into the crash dump analysis tool, the user can extract the DTI trace buffer from the crash dump and examine the records. The entire trace buffer containing the last trace records recorded by the XYZ driver is available. Using this data the user can obtain a trace showing what the driver was doing when the crash dump was taken. This allows the user to learn more about the cause of the crash.

Usage Model 2: **Troubleshooting a driver on a running system.** In this scenario, the user has encountered

```
00 01173807785:527586 0 - 02 00000000001eed86  Subchannel 0.0.4e20 reports non-I/O sc type 0001
00 01173807786:095834 2 - 02 00000000001f0962  reprobe done (rc=0, need_reprobe=0)
00 01173884042:004944 2 - 03 00000000001f7344  SenseID : UC on dev 0.0.1700, lpum 80, cnt 00
```

Figure 1: Example of the s390dbf sprintf view

a problem that is suspected to be related to one or more specific drivers. The user would like to examine the trace data just after the problem has occurred. To do so the following steps are taken:

1. Set the trace level to an appropriate level to produce the interested trace records.
2. Wait for the problem to occur, or reproduce the problem if possible.
3. Switch off tracing on the affected driver. Trace records produced just before, during, and after the problem occurred will remain in the DTI buffer.
4. Collect the trace data using a user level trace formatting tool.
5. Switch tracing back on.

By integrating DTI with systemtap additional usage modules can be realized.[2]

3.3 Debugfs

Debugfs is a minimalistic pseudo-filesystem existing mainly to provide a *namespace* that kernel facilities can hang special-purpose files of off, which in turn provides file-based access to kernel data. It provides a simple API for creating files and directories, as well as a set of ready made file operations which make it easy to create and use files that read and write primitive data types such as integers. For more complex data, it provides a means for facilities to associate custom file operations with debugfs files; in the case of relay, the exported `relay_file_operations` are associated with the debugfs files created to represent relay buffers. Despite the name, debugfs is not meant to be used only for debugging applications; it's enabled by default in many Linux distributions, and its use is encouraged especially for things that don't obviously belong in other pseudo-filesystems such as procfs or sysfs.

3.4 Relay

Relay[3] is a kernel facility designed for 'relaying' potentially large quantities of data from the kernel to the user space. Its overriding goal is to provide the shortest and cheapest possible path for a byte of data from the point it's generated in the kernel to the point it's usable by a program in user space. To accomplish this goal, it allocates a set of pages in the kernel, strings them together into a contiguous kernel address range via vmap(), and provides a set of functions designed to efficiently write data into the resulting relay buffer.

Each relay buffer is represented to user space as a file. This relay file is the abstraction used by the user space programs to retrieve the data contained in the relay buffer. The standard set of file operations allows for both read(2) and mmap(2) access to the data. These relay file operations are exported by the kernel, which allows them to be created in a pseudo-filesystem such as debugfs or procfs. In fact relay files **must be** created in one of these pseudo-filesystems in order for them to be accessible to user space programs (older versions of relay did actually include a file system called relayfs but the fs portion of the code was later subsumed by almost identical code in debugfs, and thus removed).

Relay buffers are logically subdivided into a number of equally sized *sub-buffers*. The purpose of sub-buffers is to provide the same benefits as double-buffering, but with more granularity. As data is written, it fills up sub-buffers, which are then considered ready for consumption by the user space. At the same time data is being written by the kernel into these unfinished sub-buffers, user space can be reading and releasing other finished sub-buffers. Relay channels can be configured to do double-buffereing or single-buffering if desired, or they can be configured to use large numbers of sub-buffers. A sub-buffer isn't considered readable until it's full and the next sub-buffer is entered (a sub-buffer switch). The latency between when a given event is written and the

[2] These advanced usage models will be explored in more detail in **Section 9, Integration With Other Tools**.

[3] See Documentation/filesystems/relay.txt for complete details.

time it's available to the user increases with the sub-buffer size. If sub-buffers are small, the latency is small and the amount of data that would be lost if the machine were to loses power is small. However, using small sub-buffers results in more time spent in the sub-buffer switching code (the slow path) instead of the main logging path (the fast path). Relay was designed with some reasonable middle ground in mind, efficiently buffering data implies some nontrivial amount of latency. If an application requires more immediacy, another mechanism should be considered. The assumption is that any mechanism that offers more immediacy by definition also creates more tracing overhead. The implied goal of relay is to cause as little disruption to a running system as possible.

By default, a relay buffer is created for each CPU; the combination of all per-CPU relay buffers along with associated meta-information is called a relay channel. Most of the relay API deals with the relay channels, and almost every aspect of a relay channel are configurable through the relay API.

4 Proposal for the Driver Tracing Interface

This section describes the purposed DTI architecture. The kernel API is introduced, trace record formatting and buffering are also discussed.

4.1 The DTI kernel API

The proposed DTI API can be broken in the following four major operations:

- Creating a trace handle and binding it to a relay channel.
- Writing trace records.
- Closing the channel.
- Setting the trace level.

The prototype of the API to the DTI is shown in Figure 2.

4.2 Creating the trace channel

The creation and binding of the trace handle are performed by calling `dti_register()`. When called, `dti_register()` creates a relay channel, associated data files and two additional control files in the debug file system. Upon successful completion, a `struct dti_info` pointer is returned. The caller will pass this pointer to all subsequent calls to the API. The format of `struct dti_info` is:

```
struct dti_info {
    struct dentry* root;
    struct rchan* chan;
    int    level;
    struct dentry *reset_consumed_ctrl;
    struct dentry *level_ctrl;
};
```

4.3 Writing trace records

Trace records are passed to the user using a variable length record as described in the `struct dti_event`.

```
struct dti_event {
        __u16 len;
        __u64 time;
        char data[0];
}  __attribute__ ((packed));
```

Trace record suppliers only need to supply a pointer to the data buffer containing the raw data and the length of the data. DTI places no restriction on the format of the data supplied.

4.4 Trace level

The trace level is used to control what trace records should be placed in the buffer. Suppliers of trace data provide a trace level value each time a trace record is written. The trace level value is compared to the current trace level found in `dti_info->level`. Trace records are only placed in the buffer if the supplied trace level is less than or equal to the current trace level.

The current trace level is set using `dti_set_level()` or by the user writing a new trace level to the level control file. Trace levels are defined as an integer between –1 and `DTI_LEVEL_MAX`. A value of –1 means no tracing is to be done. When a trace channel is first registered the current trace level is set to `DTI_LEVEL_DEFAULT`.

```
/**
 * dti_register: create new trace
 * @name: name of trace
 * @size_in_k: size of subbuffer in KB
 *
 * returns trace handle or NULL, if register failed.
 */
struct dti_info *dti_register(const char *name,
                  int size_in_k);

/**
 * dti_unregister: unregister trace
 * @trace: trace handle
 */
void dti_unregister(struct dti_info *trace);

/**
 * dti_printk_raw: Write formatted string to trace
 * @trace: trace handle
 * @fmt: format string
 * @...: parameters
 *
 * returns 0, if event is written. Otherwise -1.
 */
int dti_printk_raw(struct dti_info *trace, int prio, const char* fmt, ...);

/**
 * dti_event_raw: Write buffer to trace
 * @trace: trace handle
 * @prio: priority of event (the lower, the higher the priority)
 * @buf: buffer to write
 * @len: length of buffer
 *
 * returns 0, if event is written. Otherwise -1.
 */
int dti_event_raw(struct dti_info *trace, int prio, char* buf, size_t len);

/**
 * dti_set_level: set trace level
 * @trace: trace handle
 */
void dti_set_level(struct dti_info *trace, int new_level);
```

Figure 2: Prototype of the DTI API

- Three sub-buffers are shown.
- The numbers represent the trace records.
- The 1st and the 2nd trace records have already been overwritten.

Sub-buffer 1	13	14	03	04
Sub-buffer 2	05	06	07	08
Sub-buffer 3	09	10	11	12

When relay data is read, the following records are returned:

05	06	07	08	09	10	11	12	13	14

Figure 3: An example of reading relay buffers

4.5 Data buffering

DTI depends on relay to handle the data buffering. Relay arranges trace records into a fixed number of sub-buffers arraigned in a ring. Each sub-buffer may contain one or more trace records or may be unused. Records are never split across sub-buffers. As sub-buffers are filled, new records are placed in the next sub-buffer in the ring. If no unconsumed sub-buffers are available, the sub-buffer containing the oldest data is overwritten.

When trace records are read by the user using the relay read interface, the oldest complete sub-buffer returned first, then the second oldest and so on. Therefore, the trace records are returned in the exact the same order they were written. For the current (newest) sub-buffer, the trace records up to the latest written trace record is returned. We lose the rest of the trace records (the oldest ones) of the current sub-buffer. This is acceptable, if enough sub-buffers are used. This process is illustrated in Figure 3.

4.6 Picking a buffer size

When the DTI trace is registered, you supply a size-in-k value, which is the total size of each relay channel buffer:

```
dti_register(name, size_in_k)
```

DTI automatically divides size-in-k by 8 and calls the exported `__dti_register()` function:

```
__dti_register(name, subbuf_size,
n_subbufs)
```

Users normally use the `dti_register()` version which does the calculation on behalf of the user. The user is not required to understand buffer internals, however if the user wants more control over the internal sub-buffer sizes, the `__dti_register()` version is available.[4]

4.7 Record time stamps

The time field in the `struct dti_event` is generated by the DTI. Its purpose is to provide both a time reference for when the trace record was written and a tool to sequence the records chronologically. The order of the trace records in a buffer of a single CPU is guaranteed to be in chronological order. DTI creates one relay buffer for each CPU. Therefore, the user must fully read each per-CPU buffer, then order the records correctly. It is possible for records read from different per-CPU buffers to contain the same time stamp. The choice of a sufficiently high resolution timer reduces the possibility of duplicate time-stamps; if the possibility is small, it might be acceptable.

5 DTI handle API

This section describes an extension to the basic DTI API called DTI handles. The DTI handle API simplifies writing kernel code that utilizes DTI. The features provided by the DTI handle API are:

- Auto-registration
- Support for early boot time tracing

Auto-registration eliminates the need to explicitly call `dti_register()`. Both modules and built-in drivers are supported. Registration of the DTI handle is automatically performed the first time trace data is written.

Early boot time tracing allows built-in drivers to log trace data before kmalloc memory is available. Static buffers are used to hold DTI events until it is safe to setup the relay channels. The DTI handle code creates

[4] See **Section-3.4 Relay** for a summary of sub-buffer size tradeoffs to consider when choosing buffer/sub-buffer sizes.

```
#include <linux/dti.h>

static struct dti_handle my_handle

#ifdef MODULE
/* On the first event, channel will be auto-registered. */
DEFINE_DTI_HANDLE(my_handle, DRV_NAME, 4096 * 32, DTI_LEVEL_DEBUG,NULL);
#endif

#ifdef KERNEL
/*
 * Built-in drivers can optionally provide a static buffer used for
 * early tracing.
 */
static char my_buf[4096 * 4] __initdata;
DEFINE_DTI_HANDLE(my_handle, DRV_NAME, 4096 * 32, DTI_LEVEL_DEBUG,my_buf);
#endif

static int __init init_mydriver(void)
{
    ....
    INIT_DTI_HANDLE(my_handle);
    ....
}

my_driver_body(..)
{
    ....
     /* trace some events */
    dti_printk(my_handle, DTI_LEVEL_DEFAULT, format, _fmt, ## _args);
    ....
}

void cleanup_mydriver(void)
{
    ....
    CLEANUP_DTI_HANDLE(my_handle);
    ....
}

module_init(init_testdriver);
module_exit(cleanup_testdriver);
```

Figure 4: Example of using DTI handles

a postcore_initcall to switch tracing from static buffers to relay channels. All trace records written into static buffers are made available to the user interface after the postcore_initcall has run.

An example of using DTI handles is shown in Figure 4.

6 User interface

The section covers how trace data is read by the user on a running system and how tracing is controlled by the user.

6.1 File structure and control files

When a trace handle is bound, the following files are created in the traces directory of the root of the mounted debug file system.

```
dti/
    driver-name/
    data0 ... data[max-cpus]
        level
        reset_consumed
```

6.2 Retrieving trace data

One data file per CPU is created for each registered DTI trace provider. Trace records (struct dti_event) are read from the data files using a user supplied trace formatting tool. A trace formatting tool should read each per-CPU data file for a specified trace provider then arrange records according to the time stamp field of the struct dti_event.

The sequence of events normally followed when reading trace data is:

1. Switch off tracing by writing a –1 into the level file.
2. Read each of the per-CPU data files.
3. Switch tracing back on.

6.3 Trace level

The level file is used to inform the DTI provider of the level of trace records that should be placed in the data buffer. Reading the level file will return an ASCII value indicating the current level of tracing. The level can be changed by writing the ASCII value of the desired tracing level into the level file.

6.4 Reset consumed

When trace records are read, the records are marked as consumed by the relay subsystem as they are read. Therefore, subsequent reads will only return unread or new records in the buffers. If the desired behavior of a trace formatting tool is to return all records in the buffer each time the tools executed, the tool must reset the consumed value after reading all records currently in the buffer. Resetting the consumed value is performed by writing any value into the reset-consumed file.

7 Dump analysis tool support

Analysis of trace data from a crashed system is one of the most important use-cases for DTI. As such, support will be added to the crash and lcrash dump analysis tools enabling those tools to extract and make use of the DTI trace buffers and related trace information from a crashed system image.

7.1 Retrieving trace data from a crash dump

When a DTI trace is registered through dti_register(), a text name is specified as one of the parameters. This string not only identifies the trace to the user, but is also effectively concatenated with a prefix string, unlikely to be used by a user, in order to make the complete identifying string easily locatable in a memory dump for example if a trace name is given as "mydriver" by the user, the string the dump tool would use to find the corresponding dti_info struct would be __DTI__mydriver. When this string is located, it's a straightforward exercise to locate the associated relay channel and its buffers. For example, assuming a simplified dti_info struct:

```
struct dti_info
{
  struct rchan *chan;
  char dti[7] = "__DTI__";
  char name[DTI_TRACENAME_LEN];
  .
  .
  .
}
```

The dump tool would locate the beginning of the dti[] array, and subtracting the size of a pointer would find the pointer to the struct rchan:

```
struct rchan
{
  size_t subbuf_size;
  size_t n_subbufs;
  struct rchan_buf[NR_CPUS];
  .
  .
  .
}
```

From the pointer to the rchan, you can locate each of the buffers that make up the channel and its size.

```
struct rchan_buf
{
  /* start of buffer */
  void *start;
  /* start of current sub-buffer */
  void *data;
  /* write offset into the current
     sub-buffer */
  size_t offset;
  /* number of sub-buffers
     consumed */
  size_t consumed;
  .
  .
  .
}
```

From this information, you can extract all the available data from each buffer and send it to the post-processing tool to be combined and sorted as usual.

8 Sample implementations

8.1 Port of the S390 dbf

Most of the s390 device drivers currently use the s390dbf. Examples are:

- ZFCP: SCSI host adapter device driver
- QETH: Ethernet network device driver
- DASD: s390 harddisk driver
- TAPE: Driver for 3480/90 and 3590/92 channel attached tape devices

s390dbf API	DTI API
`debug_register()`	`dti_register()`
`debug_unregister()`	`dti_unregister()`
`debug_event()`	`dti_event_raw()`
`debug_sprintf_event()`	`dti_printk_raw()`
`debug_set_level()`	`dti_set_level()`

Table 1: Mapping of the s390dbf API to the DTI API

All the users of the s390dbf API can be quite easily converted to use the new DTI API. Table 1 shows functions that can be mapped directly.

There is no DTI equivalent for the s390dbf exception calls. Those must be replaced by the appropriate DTI event functions. The exception functionality used to switch debug areas is not frequently used by the s390 device drivers. Therefore it is acceptable to remove this functionality in order to keep the API small and simple.

The functions `debug_register_view()` and `debug_unregister_view()` are not needed any more, since formatting of DTI traces is done in the user space.

Currently the s390 ZFCP device driver uses non-default self-defined s390dbf views. For that driver, it is necessary to implement a user space tool with the same formatting functionality as the ZFCP specific s390dbf view.

Some details of the port have not yet been resolved.

- How should the number of pages value used by `debug_register` be mapped to the buffer size used in `dti_register()`?
- Is an equivalent for `debug_stop_all()` needed?

8.2 Port of some other drivers

There are currently a large number of custom logging APIs in the kernel, each mainly restricted to logging formatted debugging string data related to a particular driver or subsystem. Most of them are a variation on `#define DPRINTK(x...) printk(x...)`.

These can easy be ported to DTI using the DTI handle API. However, DTI's strength is focused on continuous tracing to a buffer which can be retrieved when necessary rather than the continuous logging implemented by these special-purpose facilities.

There are, however, a handful of continuous tracing facilities similar to DTI in the kernel, varying from very minimalistic to fairly full-featured. Included in this category is the current s390dbf facility, which is presently the most advanced. Briefly examined are some of the others that tracing facilities that would be candidates for porting to the proposed DTI API.

- `drivers/scsi/mesh.c` provides the `dlog()` function for logging formatted records. It keeps data in a ring of structs. The dump function printk's the whole buffer.
- `drivers/net/wan/lmc/lmc_debug.h` provides `LMC_EVENT_LOG()` which logs two u32 args along with an event number and jiffies.
- `drivers/char/ip2` provides `ip2trace()` which is used to trace any number of longs into a buffer containing 1000 longs. It provides a read(2) interface to read the data.
- `drivers/isdn/hardware/eicon/debug.c` provides an extensive API and set of functions used to log and maintain a queue of debug messages.
- `fs/xfs/support/ktrace.c` provides a simple yet extensive API for tracing the xfs file system. The main logging function is `ktrace_enter()`, which allows up to 16 values to be logged per entry into a buffer containing 64 ktrace entries.

9 Integration of other tools

DTI tracing is of course, useful in its own right. Combined with a trace analysis tool such as SystemTap[5] it can become an even more powerful tool. A DTI trace can be used to see with great detail exactly what happens within a particular driver over a given time interval, but it doesn't have any other context associated with it, such as system calls or interrupt activity that may have triggered activity within the driver. In many cases, it would be extremely useful to have such associated data available for analysis. DTI's strength is its data-gathering functionality. Using Systemtap, DTI's functionality can be extended by adding context to the data, perform time analysis of the data being gathered or maintain a summary information about the trace. For example, if a certain pattern of interest only occurs intermittently, the user could detect it and either halt the trace, preserving the events that led up to it, or alert the user of the condition. In addition, SystemTap can be used to gather aggregated summaries of the data over long periods of time (longer than the limited size of a relay buffer would allow) to get an overall picture of activity with respect to the driver and associated context.

The DTI tapset is being developed to allow SystemTap to put kprobes on the high-level DTI tracing functions, which makes all data passing through them accessible to SystemTap. Note that doing this in no way affects the normal flow of DTI events; the only additional effect is the probe effect, which means that each event recorded in this way incurs a penalty equal to the time required to fire the probe and run the SystemTap handler.

Systemtap can also be used to dynamically place new probe points into a driver. To do this Systemtap places a kprobe in the driver where a trace point is to be added and all data available at the probe point is fed back into the DTI data stream. The DTI post-processing tools can then be used to format and display this external data along with the normal DTI trace output. To do this, users make use of the 'DTI-control' tapset, which allows SystemTap scripts to control and log data to DTI via the standard DTI kernel interface.

10 Conclusion

Customers of s390 systems demand very high reliability and quick turnaround time for bug fixes. Since its introduction early in the history of Linux on the s390, the s390 Debug Facility has proven itself as an invaluable tool for meeting customers' reliability expectations. The ability to analyze trace data in a crash dump and perform first fault analysis is key to the s390dfb's success. The question has been posed, "Why reinvent functionality that already exists?" The answer is simple: DTI intends to exploit the s390db model and bring this technology to all Linux platforms. Service organizations gain not only by the additions of DTI functionality but by the uniformity of a tracing infrastructure between all platforms. In

[5] `http://sourceware.org/systemtap`

addition, DTI utilizes the existing relay subsystem that did not exist when s390dbf was written. Therefore, DTI can be implemented with a much smaller footprint than the s390dbf.

s390 drivers provide the perfect sandbox for porting drivers to the DTI and testing its implementation. We plan to pursue this effort as well as encouraging other driver developers to adopt DTI.

Legal statement

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM, IBM (logo), e-business (logo), pSeries, e (logo) server, and xSeries are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds.

Other company, product, and service names may be trademarks or service marks of others.