A Universal Dynamic Trace for Linux and other Operating Systems

Richard Moore - IBM, Linux Technology Centre - richardj_moore@uk.ibm.com

Abstract

Dynamic Probes (DProbes) from **IBM [*]** is a generic and pervasive system debugging facility that will operate under the most extreme software conditions with minimal system disruption. It permits debugging of some of the most difficult types of software problem especially those encountered in a production environment that will not readily re-create. It is also an invaluable aid for the developer who has to debug parts of the operating system inaccessible to other technologies. **DProbes** is a front-end enabler for other debugging technologies, such as crash and core dumps and kernel/user debuggers. It is designed to operate with minimal dependence on the operating system, which affords it the possibility of being ported to other operating systems, especially **UNIX** [**] variants, but not limited to **UNIX** as it originated conceptually from **Dynamic Trace** under **OS/2** [*]. This paper describes the latest developments of the **DProbes** project in particular it use as a tracing tool with the **Linux Trace Toolkit** project from **Opersys [**].** System dependencies are discussed with an emphasis on portability to other **Linux** H/W platforms as well as other operating systems.

1. Introduction

Dynamic Probes (DProbes) for Linux <1> was first released in August 2000 and presented at the Annual Linux Showcase in October 2000 <2>. The original functionality was essentially that of an automated kernel debugger. Since then DProbes has been extended considerably. It now interfaces with a number of external debugging agents, for example: The Kernel Debugger <3> and Kernel Crash Dump <4> facilities from Silicon Graphics Inc. (SGI) [**]; the standard user-space core dump and syslog facilities within Linux and also the Linux Trace Toolkit <5> from Opersys.

The major topics discussed in this paper are:

- Detailed implementation aspects of DProbes that relate to its use as an agent for trace instrumentation under the Linux operating system running on the Intel 32-bit architecture (IA32) <8>.
- Portability considerations across other operating systems running under the Intel 32-bit architecture in particular UNIX-like operating systems.
- Portability to other processor architectures.

Essentially DProbes has become a driver or **enabler for other debugging technologies**. Its enabling capability derives from the following key characteristics:

1. There is a mechanism for intercepting execution at arbitrary code locations - this is the **probepoint** mechanism.
2. Each probepoint has an associated **probe handler** that allows specific actions to be taken. This is implemented using a low-level Reverse Polish Notation (RPN) language that gives access to kernel and user space memory and to the processor's registers[1].
3. A probe handler terminates in one of three ways:
    i. By returning to the probed code seamlessly.
    ii. By returning to the probed code via a logging dæmon. A temporary logging buffer is made available for this purpose. This characteristic is exploited to provide a means of instrumenting a module with **tracepoints**.
    iii. By transferring control to an external debugging facility having first removed the probepoint. Whether or not the original code will continue execution is a function of the external facility.

The efficacy of DProbes is further enhanced by the following three design criteria:

1. There is no required interactive user interface for the probe handler[2]. This is intentional - it minimizes the dependency of the probe handler on system interfaces and resources. Thus the probe handler is designed to run as a self-contained interrupt handler. The RPN command interpreter provides recovery form potential fatal errors without reference to operating system facilities. This criterion gives DProbes its **universality** since there are very few restrictions on where a probepoint may be placed and when the probe handler may execute. In fact, probepoints are only restricted from being placed in the code path of the probe handler. If such a probepoint were to be defined, DProbes would detect it and silently remove it. Probepoints may therefore be placed in code that runs at task time, interrupt time or during a context switch.

2. The second design criterion was to align a probepoint with a module rather than a storage location. Note that the watchpoint extension, which is described under 7. Dynamic Probes Recent Extensions, deviates from this criterion for reasons explained thereunder. By aligning a probepoint with a module, or to be more precise an offset into a module, the probe becomes independent of incidental circumstances that relate to a module's installation in memory and also the processes under which that module is executing. Thus, code that is shared between processes at different virtual addresses (for example a Linux shared library) has location-independent and process-independent probe definitions. This criterion gives DProbes its **independence,** since it makes it possible to describe a probe:

   i. Independently of an operating system's implementation of module management (clearly the internal implementation needs to understand this);

   ii. In canonical terms that relate to a programmer's view of his/her module which are: independent of whether a module is loaded at the time the probe is defined; and independent of any particular process under which that module executes.

3. Probepoints are *inserted* into module code paths without the need for source code modifications to that module. Furthermore they may be inserted into any loaded and running code (kernel or user space) or code that is paged out or modules not yet loaded. This mechanism has been described in <2>. In summary, the instruction at the probe location is overlaid with a trapping instruction - under Intel 32-bit architecture (IA32) the **int3** instruction is chosen. The original instruction is either single-stepped or emulated after the probe handler executes. This particular criterion gives DProbes its **dynamic** characteristic. The dynamism refers to its ability to instrument a module with probes *on the fly* so to speak. Thus there is absolutely no performance penalty when a probepoint is inactive.

These characteristics therefore provide an elementary universal dynamic tracing capability, which have been further extended in both universality and dynamism by recent enhancements - see the section: 7. Dynamic Probes Recent Extensions for details.

2. DProbes as a Tracing Mechanism

This section discusses the implementation details of DProbes as a tracing mechanism in detail. We describe first the internal mechanism of the Dynamic Probes Event Handler (DPEH) that enables it to be used as a tracing agent.

DPEH internal details:

DProbes provides various working storage elements for use by the probe handler:

1. Local variable array.
2. Global variable array.
3. A per-processor log buffer.

The latter is intended for use as a staging area for building a trace or log record to be passed synchronously to a logging or tracing dæmon external to DProbes. One log buffer is permanently allocated per processor but the data in each buffer persists only for the duration that a probe handler is active on its respective processor. The RPN command interpreter maintains an internal pointer to the next available location in the buffer, which is reset to the beginning of the buffer on entry to the probe handler. Data is thus always accumulated monotonically and discarded on exit from the probe handler.

The buffer is populated using the **log** class of RPN instructions. These are defined in two categories: those that copy data directly from the RPN stack and those that use the RPN stack to specify data to be copied from system memory.

The direct category comprises three instructions in the IA32 implementation:

log b,<n>	Log byte
log w,<n>	Log word
log d,<n>	Log double-word

Pop <n> elements from the RPN stack and from each, copy the least significant byte (8-bit integer), word (16-bit integer) or double-word to the log buffer.

The indirect category has four members in the IA32 implementation. Each operates by popping an address followed by a length from the RPN stack. Data, for that length, at that address, is copied to the log buffer. However, before data is copied, it is appended with a 3-byte prefix that contains a token byte and a length word. The length refers to the length of data that follows the prefix and the token byte to type of data. The feature enables:

a. A trace or log record to contain variable length data such as arrays whose length is determined dynamically.

b. A formatting utility to operate using a fixed template for variable length data.

The token byte values are defined as follows:

0	binary data logged successfully of length specified by the prefix length word.
1	(ASCII) string data logged successfully of maximum length specified by the prefix length word. The actual length of the data may be less if a terminating NULL byte is encountered within the prefix length. This allows data of arbitrary lengths to be capped especially in cases where string data has been corrupted.
-1	a fault occurred accessing the data using a flat address. The prefix length is set to 4 and the variable data contains only the (flat) address that caused the fault.
-2	a fault occurred accessing the data using an invalid selector. The prefix length is set to 4 and the variable data contains only the selector for the segment generating the fault.

Note: the latter two tokens may occur in circumstances where the data address was valid. Since the RPN probe handler executes, essentially as an interrupt handler, with minimal access to system facilities it will not be able to recover from otherwise recoverable faults. This is a trade-off between the universality and flexibility of DProbes. See 3. DProbes Event Handler Processing for a further discussion on how such conditions are handled.

Under the IA32 implementation there are 4 RPN instructions use for logging data in memory:

log mrf	Log memory range from flat address.
log mrs	Log memory range from segmented address.

Pop a flat address (**log mrf**) or a 16-bit offset then a 16-bit selector (**log mrs**) from the RPN stack. Then pop a length. Reserve space for the 3-byte prefix in the log buffer. Copy data from the address specified for the length specified to the log buffer. If a fault is generated then set the prefix token to -1, the length to 4 and store the fault address. If no fault is generated then set the prefix with a 0 token and length of data logged.

log arf	Log ASCII range from flat address.
log ars	Log ASCII range from segmented address.

Pop a flat address (**log arf**) or a 16-bit offset then a 16-bit selector (**log ars**) from the RPN stack. Then pop a length. Reserve space for the 3-byte prefix in the log buffer. Copy data from the address specified up to the length specified or until a NULL terminator byte has been copied. If a fault is generated then set the prefix token to -2, the length to 4 and store the fault address. If no fault is generated then set the prefix with a 1 token and maximum length value popped from the RPN stack.

3. DProbes Event Handler Processing

We turn now to considerations concerning back-end probe event handler processing.

As described in <2>, the DProbes Event Handler (DPEH) needs to execute the original instruction that was replaced with a breakpoint. It does this by single-stepping the original instruction *in situ* with interrupts disabled[3]. If that instruction faults then we require the operating system to recover and retry the instruction. However, one does not normally wish to have multiple trace records generated for each retry execution of an instruction, especially when that instruction eventually succeeds and will thus appear to the underlying program to have executed only once, and with success. This is achieved by delaying the call to the tracing dæmon until after the original instruction has completed single-step. If a fault is generated then the dæmon is not called and the log buffer is reset. Furthermore the DPEH reinstates the probepoint - **int3** instruction under IA32 - and resets interrupt status, saved by the processor when the fault was generated, to indicate the status prior to execution of the probepoint. Finally it returns to the system fault handler to allow normal fault processing to occur. If the system retries the faulting instruction it will unwittingly retry the probepoint instruction. The DPEH will therefore be called for each retry. Only on successful execution of the original instruction will the trace dæmon be called.

It is a requirement for the probe handler to be re-executed for each retry of a faulting instruction. This is because it is quite possible, in the case of a page fault, that the data causing the instruction to fault is also accessed by the probe handler for copying into the log buffer. Only on successful execution of the original instruction would the trace record in this case be complete.

Some instructions generate faults for non-error reasons, for example IA32 **bounds** instruction. For such instructions it would be desirable to log a trace record despite a fault being generated on single-step. This is now possible by means of the **logonfault** control statement, which is specified in the header of the RPN file. This feature was added recently to DProbes - see 7, Dynamic Probes Recent Extensions below.

4. DPEH Performance Implications

We have made an initial study of the performance overhead of a probepoint. A more comprehensive performance evaluation is future work. The first set of results are quantitative observations made under the Linux 2.2.12 kernel. We also present some qualitative results taken from real-life usage under OS/2. The conclusions from these OS/2 examples indicate that under most conditions the impact of a probepoint is negligible when active. We concern ourselves only with measuring the effect of the active probepoint, since for inactive probepoints there is no alteration to the code path and therefore a zero overhead.

We estimated the overhead of the DPEH using a 90 MHz Pentium[**] processor[4]. Five experiments were performed:

1. To obtain a base measurement of the time taken to execute a sequence of the following three single cycle instructions in a loop:

   ```
   loop: dec eax
         nop
         jnz loop
   ```

2. To test a null probe handler with only the **abort**[5] RPN instruction and with the probe placed on the **dec eax** instruction. Here **dec** is single-stepped by the DPEH.

3. Using the same probe handler but the probe placed on the **nop** instruction. Here **nop** in emulated by the DPEH.

4. Using same probe location as 2 but a single **push eax**[6] RPN instruction added to the probe handler.

5. The same probe location as 2 but a single **exit**[7] RPN instruction in the probe handler.

The results were as follows:

1. One iteration of the three-instruction loop averaged 30ns, each instruction approximately 10ns .

2. One iteration of the loop averaged 16µs. Therefore the minimum overhead of the DPEH is approximately 16µs.

3. One iteration of the loop averaged 8µs. Therefore the cost of the DPEH back-end single-step processing accounts for half the overhead per probe.

4. One iteration of the loop averaged 16µs. **push eax** therefore has a negligible effect. One might reasonably assume most register and memory based RPN instructions are of a similar overhead.

5. One iteration of the loop averaged 200µs. Most of this is the cost of a **printk**, which is the default logging method (see 5. The Trace Dæmon Interface below for details on how **printk** in invoked).

Taken at face value the minimum overhead of the DPEH appears to be of the order of 10^3. This would certainly be a valid perception if a probe were placed in a tight CPU-bound loop. However, in most applications of DProbes the average number of instructions executed between consecutive executions of the same probepoint outweighs any overhead imposed by the DPEH. This is illustrated by the following qualitative results taken from real-life uses of DProbes (actually Dynamic Trace) under OS/2:

1. Tracepoints[8] on every kernel API entry and exit (circa 500 tracepoints).
   The user perception varies from unnoticeable to very slight depending on work load. The system is useable and performs within acceptable norms.

2. Tracepoints on entry and exit to the process context switching code with page table data logged on entry and exit.
   No noticeable overhead.

3. Tracepoints on every OS/2 Presentation Manager [*] API entry and exit (circa 500 tracepoints).
   A noticeable slowing of GUI response. The GUI was useable.

4. Tracepoints on entry to page allocation, page de-allocation and page fault handling routines in the OS/2 page manager.
   No noticeable overhead.

5. Tracepoints on 4000 internal kernel routines.
   Very noticeable, however the system was still useable.

Conclusions

While the cost of a probe is not cheap it can be considerably reduced by placing the probe on an emulated instruction such as **nop**. It can also be reduced by judicious use of logging by employing conditional logic in the probe handler to avoid unnecessary log events. But foremost, the practical use of DProbes finds probepoints being placed in code paths with a relatively long mean time to iterate. Under these circumstances the overhead is negligible.

5. The Trace Dæmon Interface

The generic requirements for a trace dæmon interface are:

1. To provide a logging API capable of being called from kernel space, while interrupts are disabled, from both a task-time and interrupt-time context.

2. To allow binary data of an arbitrary length to be logged and identified as originating from DProbes.

A number of candidates satisfy these requirements. The default behavior is to invoke the **klog** dæmon via **printk**. Other options include directing output through a dedicated asynchronous communications port (**com1** or **com2**). Strictly speaking, using a communications port doesn't necessarily invoke a dæmon unless one thinks of the monitoring system connected to the system running DProbes as a dæmon. And finally, a local tracing dæmon can be invoked to record the log buffer. Use of this option requires a degree of conformance between both DProbes and the tracing facility. We have chosen to use the Linux Trace Toolkit from Opersys <5> as an initial implementation. We will describe a little later a generic interface that is possible to implement by using the Generalised Kernel Hook Interface <6> mechanism.

Logging to the Communications ports or klog:

Logging to both the **com1** and **com2** communications ports and **klog** involves converting the log data to an ASCII string of pairs of hexadecimal characters and outputting that to the respective medium. Prior to this we format a record header that contains both constant information and some optional entities that are common to all tracepoints. The most generalised form of the trace header template is as follows:

```
"DProbes(%d,%d) cpu=%d name=%s pid=%d
uid=%d cs=%x eip=%08lx ss=%x esp=%08lx
tsc=%08lx:%08lx\n"
```

Other than the **DProbes(...)** text item, every other item is optionally present. All but **cpu** are selectable by the user through parameter switched to the **dprobes** command; **cpu** is activated automatically when DProbes is run from a multi-processor system.

The meaning of each constituent header item is as follows:

DProbes(%d,%d)
Displays the major and minor code that identifies the probepoint. Each probepoint has assigned a major and minor identifier. These are not required to be unique, but by convention are chosen to indicate a unique type of probe, for example the exit point(s) of a particular routine. Major and minor codes are intended to be used by a generalised formatter to identify a unique formatting template. See 6. Trace Formatting Interface below

cpu=%d
Displays the processor id on which the probe was executed. This is suppressed on uniprocessor systems and always displayed on multi-processor systems.

name=%s
Displays the process name taken from the current task structure when the probe was executed.

pid=%d
Displays the process id taken from the current task structure when the probe was executed.

uid=%d
Displays the user id name taken from the current task structure when the probe was executed.

cs=%x eip=%08lx

Displays the CS and EIP registers at the probe location. This is sometimes useful in distinguishing individuals of a group of similarly formatted and therefore identical major and minor coded probes. For example, multiple return points from a function.

ss=%x esp=%08lx

Displays the SS and ESP register values when the probe was executed. This can give an indication of the nesting level of a subroutine.

tsc=%08lx:%08lx

Displays the high resolution processor time-stamp counter in seconds and micro-seconds.

The remaining data is output as an ASCII string of hexadecimal characters.

Logging to a Trace Daemon

We chose to use the Linux Trace Toolkit (LTT) <5> from Opersys as the trace recording dæmon. It provides the usual post-processing, formatting and analysis features as well as a dæmon that manages the a kernel space trace buffer and a mechanism for off-loading the trace buffer to disk. But most importantly, the Linux Trace Toolkit was conceived as a kernel based static[9] tracing mechanism, capable for having tracepoints placed in both interrupt handlers and code that runs with interrupts disabled. In other words the conditions under which the DPEH executes. We were able extend the Linux Trace Toolkit to provide an kernel programming interface that allows data to be logged of a an arbitrary length.

The KPI interface to Linux Trace Toolkit's Raw Data interface is show below:

```
struct trace_raw {
     uint32_t  id;  /* Event ID */
     uint32_t  DataSize; /* Size of data
                      recorded by event */
     void*     Data;     /* Data recorded by
                         event */
}

#define TRACE_RAW(ID, LEN, DATA) \
     do { \
          struct trace_raw raw_event; \
          raw_event.id = id; \
          raw_event.DataSize = LEN; \
          raw_event.Data = DATA; \
```

event

This an event identifier defined by LTT. It signifies a binary data record, the format of which is undisclosed to LTT.

id

This a module identifier returned by LTT when tracepoints for a given module are activated. DProbes calls the LTT **trace_create_event()** routine when it inserts tracepoint for a given module. This enables LTT to correlate events with a module for the purposes of event analysis. Note: DProbes will call LTT **trace_destroy_event()** routine when tracepoints for a module are removed.

DataSize

This is the overall size of the trace record (flags + + log buffer content).

Data

This is a pointer to the trace record (flags + header + log buffer content).

The logged data is further structured with a header followed by the data from the log buffer. The header comprises a flag double-word followed by one or more binary data items concatenated together. The presence of an item is signified by its corresponding flag bit being set. The following table shows the format of each header item and its corresponding flag setting in the order they appear in the header:

#	flag	type	description
1	0x0001	uint32	major
2	0x0002	uint32	minor
3	0x0004	uint32	cpu
4	0x0008	uint32	pid
5	0x0010	uint32	uid
6	0x0020	uint32	cs
7	0x0040	uint32	eip
8	0x0080	uint32	ss
9	0x0100	uint32	esp
10	0x0200	uint64	tsc
11	0x0400	string	process name

This implementation is specific to LTT, but may be readily adapted to other dæmons either by requiring that they support the three interfaces for creating, destroying and logging an event.

Generalised Kernel Hook Interface

The disadvantage of the implementation just described is that DProbes needs to be built for use with LTT and LTT needs to be present in the system before DProbes loads in order to resolve the external references to the three interfaces. We can avoid this problem by using the Generalised Kernel Hook Interface <6> to define hook exit points within DProbes for the three interfaces. An arbitrary trace dæmon would register and arm exit routines for these three hooks when the dæmon loads or is instructed to do so. Because the state of activation of a GKHI hook is transparent to DProbes, it would execute code paths that call the three interfaces (now hooks) without regard to whether a recipient dæmon had armed them. The equivalent hook exit points for each of the three API calls is coded as follows:

trace_event(event, &event_struc);
GKHOOK_2VAR(GKHOOK_DPROBES_LOG_EVENT, event, &event_struc);

event=trace_create_event(name, format, desc);
GKHOOK_4VAR(GKHOOK_DPROBES_CREATE_EVENT,&event, &name, &format, &desc);

rc=trace_destroy_event(event);
GKHOOK_2VAR(GKHOOK_DPROBES_DESTROY_EVENT,&rc, event);

DProbes would notify GKHI of the existence of these three hooks during initialisation by calling **GKH_identify**.

6. Trace Formatting Interface

Clearly, a hexidecimal format for the trace record is not the most user friendly. Therefore we have proposed a formatting utility in the form of a set of shared library routines that may be called to format individual trace records. The unformatted binary trace record is passed to the formatter and a pointer to the formatted trace record is returned.

The formatter uses text templates with place-holders to format the raw data. For efficiency, templates are cached in memory. The formatting library provides two additional subroutine calls:

1. **initialise**, where essentially the template directory file is opened, loaded and closed.
2. **terminate**, where any cached templates are freed.

Note: these two interfaces may be called sequentially, in reverse order to allow templates to be re-read from disk following an update.

Formatting Template Structure

The template syntax is an extension and simplification of that employed by the OS/2 Trace Formatter, which is a natural thing to do since Dynamic Probes also owes its origin to OS/2's Dynamic Trace facility <7>. This scheme is based on a **printf**-like formatting template. But as discussed below, we have a requirement to format arrays and binary data (essentially an array of bytes) whose number of elements is only determined at the time a trace record is created. This requirement necessitates deviation from a simple **printf** template.

By convention a unique major code is assigned per module. Each unique trace record format for a module is assigned a unique minor code within the major code. This allows us to employ one formatting template file per major code. A template directory is employed to cross-reference major code to template file name. The template file needs only to identify minor code to delimit each template, however, for sanity purposes the major code is coded at the head of the file.
Comments are allowed using c-style comment syntax.

Each formatting statement is of the form
keyword=<value>

Numeric values are allowed to be expressed in decimal and hexadecimal using c-notation.

Strings are quoted using c-notation.

The first statement of the file is:
major=<major code>

Subsequent statements will follow the format:
minor=<minor code>[,]
desc=<"descriptive header text">[,]
fmt=<"template 1">[,]
fmt=<"template 2">[,]
.
.
.

End of file or the next **minor** keyword delimits the end of the previous template.

The **desc** statement serves to provide a static text description of the trace event.

Major, minor and **desc** are mandatory, **fmt** is optional, however if **minor** is omitted then only default formatting will be performed. The data will be treated as binary and formatted in dump format displaying offsets, hexadecimal and ASCII.

The **fmt** statements are used to supply template information for formatting user data in the trace record.

In general any alphanumeric character found in the **fmt** statement is treated as literal text and copied directly to the output buffer. Escape control characters **\n** and **\t** are supported. In general the last pair of characters in a sequence of **fmt** statements will be **\n**, however the formatter will always generate an additional new-line at the end of a new trace record.

Multiple **fmt** statements for the same minor code are concatenated by the formatter, so the user must supply necessary spacing and new-line characters if the formatted data is to span more than one line. Place holders for data to be extracted and formatted within the template is signified by a sequence that is prefixed with a **%** character. Multi-byte control sequences are terminated by any non-numeric character, since in a multi-byte control sequence the trailing characters are numeric.

The following control sequences may be specified:

%<n>c - format n-bytes as an ASCII characters. If the character is in the range 0x20-0x7f then format the ASCII equivalent character, otherwise substitute a period.

%<n>d - format and n-byte decimal integer with leading zeros removed.

%<n>f - format an n-byte floating point numeric with leading zeros removed.

%<n>i - skip n bytes in the unformatted data buffer.

% p - skip the three-byte prefix for variable length data, see the description of the logging RPN commands under 2. DProbes as a Tracing Mechanism above. This is used in combination with most other controls by placing then after **p**. Controls **u** and **r** are excluded from use with **p**.

% r - skip the three byte prefix for variable length data, but use it as a repetition control, see below. This is used with other controls or a complex expression following.

% s - format an ASCII string up to the length specified by the **% p** prefix, or until a null terminator is encountered. If **%p** is not specified then **% s** formats a string until a null is encountered.

%<n>u - format an n-byte unsigned decimal integer with leading zeros removed.

%<n>x - format and n-byte hexadecimal integer including leading zeros.

% z - format the remainder of the trace record in dump format (offset, 0x20 hexadecimal bytes separated by spaces and ASCII equivalent for each 0x20 bytes, repeated for each 0x20 bytes - one per line).

```
+00000000 21 22 23 24 25 26 27 28 20 c4 a8
fe ae ef ff bb *abcdefgh .......*
+00000020 21 22 23 24 25 26 27 28 20 c4 a8
fe ae ef ff bb *abcdefgh .......*
+00000040 21 22 23 24 25 26
              *abcdef*
```

(- begins a complex expression - see below
) - ends a complex expression - see below

Where a **<n>** qualifier is allowed then its omission defaults to 1.

Processing the 3-byte prefix
% p causes the formatter to skip over the prefix, noting the code and length. If an error is indicated an error message is formatted.

If **% s** follows **% p** and the code is 0x01 then data is formatted up to the first null character or until the length is exhausted.

If **% s** follows **% p** and the code is 0x00 then data is formatted up to the first null character and any remaining data up to the value of the length is skipped.

If any other control follows **% p** then that data is formatted according to the following control, having skipped the prefix (the error code being checked first).

% p may be combined with any control other then **% r** and **% u**.

% r is used to process the prefix in a similar way to **% p**, except in this case it uses the prefix to repeat the control sequence that follows until data of the length specified by the prefix is formatted. **% r** may be combined with any control though it seldom makes sense to combine it with **%p, %s** in simple formatting expressions.

When controls are combined only one **%** is specified. For example:
%ps - causes a prefixed string to be processed.

When two data items are to be concatenated then two **%** signs are needed. For example:

%4us - formats a 4-byte unsigned decimal integer suffixed with a character s, whereas
%4u%s - formats a 4-byte unsigned decimal integer concatenated to a zero terminated string.

%r may be followed by a left parenthesis **(** to form a complex formatting expression, which is completed with a right parenthesis **)**. This device allows arrays of structures to be formatted. For example an array for which each entry contained two double-words called "function" and "return code" would be formatted using:

%r(function=0x%2x return code=0x%2x\n)

The result would be (for a length value of 12 in the prefix):

```
Function=0x0000 return code=0x0000
Function=0x0000 return code=0x0003
Function=0x0002 return code=0x0000
```

A more complex example where the array is a table pointers to strings could be formatted using:

%r(pointer=0x%4x, string='%ps'\n)

The result would be:

```
pointer=0x801234455, string='this is an example string'
pointer=0x802234455, string='this is another example
string'
```

Within a complex expression the **%** must be used to prefix groups of controls.
To format a literal **%, (** or **)** character then an additional prefix **%** is required. For example:

%%	results in	**%**
%(	results in	**(**
%)	results in	**)**

Note: there is scope for extending this scheme to cope with formatting bit masks and conditional formatting and this is something we plan to do.

7. Dynamic Probes Recent Extensions

Since its original release, Dynamic Probes has been enhanced with a number of new features which are relevant to tracing. These are briefly described below:

Watchpoint Probes

This innovation defines a new class of probe that exploits the hardware watchpoint[10] architecture. Watchpoints are specified by watch-type, which under IA32 may be Read, Write, Execute or IO; and address range. Watchpoints are global and not aligned with any particular module, however symbolic expressions are permitted in the specification of a watchpoint address. This capability gives DProbes its ability to trace memory accesses.

Logonfault

This allows the option of logging the contents of a log buffer whether or not the instruction at the tracepoint generates an exception during single-step. If the operating system retries the instruction then multiple events will be logged. This was introduced to handle two circumstances:

1. where instructions such as **bounds** generate exceptions as part of normal execution and the exception is not subject to seamless recovery by the operating system.

2. when a probe is used for monitoring program efficiency. For example, by logging all attempted executions of an instruction that is capable of generating a page-fault. By this means one may glean an insight into the effects of a particular code path on demand paging.

In both cases it is acceptable log each execution of the probed instruction whether or not it is for recovery purposes.

Probe handler exception handling

This capability allows an RPN probe handler to specify a label from which execution will continue should a fault occur when processing a **log** instruction. The 3-byte prefix is optionally generated with the error code depending on the definition of the exception handler. Interpretation of the RPN probe handler is allowed to continue.

Call Kmod

This allows an open-ended extension to the RPN command set, by providing a hook for which any kernel module may register. The **call kmod** RPN instruction will give control to the hook exit routine. GKHI is used to implement this interface.

8. Porting Considerations:

Because DProbes relies on few operating system interfaces it is relatively easy to port to other operating systems, especially of the UNIX variety. Furthermore it is structured in a way that enables it to be ported to other architectures besides IA32. IBM is currently working on ports to the zSeries (31-bit and 64-bit) [*] and Intel 64-bit <8> architectures.

Porting to Linux on other processor platforms
The following are the key items to be translated when considering a port to another processor architecture:

Integer size

The global and local variable array element size is set to the integer size (in multiples of 8). So also the element size of the RPN stack. All these dependencies are tied to a single **#define** definition.

RPN instruction set

References to processor registers need to be mapped to the new architecture. Each register push instruction is actually an alias for the single instruction **push r,<n>.** The aliases are implemented by the **dprobes** command from a table that cross-references register to register number.

The push byte, word and double-word set may need to be extended to include a quad-word (64-bit). The will need to be implemented to produce the correct results for the particular endian characteristic of the processor.

It is unlikely that a probe handler written for one architecture would work without modification for another. However this can be addressed by using a high-level language interface for probe handler definitions. This would avoid low-level CPU based constructs and have a good chance of being architecturally independent.[11]

Probepoint implementation

Probepoints are implemented trapping instruction breakpoints. The processor architecture must provide an instruction that can be stored atomically and will case a privilege-level switch. For example, **SVC 255** serves this purpose for IBM zSeries processors. The interrupt handler for the breakpoint instruction will need to be hooked by the DPEH.

Single-step

The original instruction at the probepoint needs to be single-stepped. Such a mechanism must therefore exits for use under software control. Use of the hardware watchpoint mechanism may be needed to implement this. Under IBM zSeries, one would use the Program Event Recording (PER) facility. If no inherent single-step capability exists then use of additional breakpoint instructions will be required - this however is an imperfect solution which may prohibit the specification of probepoints on jump or call instructions.

Processor exceptions

All processor exceptions that are generated through normal instruction execution need to be intercepted as part of the single-step back-end processing. Additionally the sequence from breakpoint interrupt through to single-step interrupt needs to be conducted with interrupts disabled in order:

1. to preserve event sequences where an interrupt occurs during the processing of a probe event
2. to avoid difficulties that arise with recursion through the DPEH.

Under Linux for IA32 this required both the exception 1 and exception 3 trap gates to be converted to interrupt gates.

Processor serialisation

Under a multi-processor environment the single-step optionally needs to be executed while other processors suspend execution. If this facility cannot be guaranteed then the **-stopcpus** switch of the **dprobes** command will not be supportable.

Instruction cache serialisation

Because instructions of loaded modules are dynamically altered, serialisation of the instruction pre-fectch cache may need to be performed. Under Linux the **flush_icache** operating system call achieves this.

Watchpoint implementation

This is more complex and difficult to generalise. In the worst case scenario, watchpoint probe support will have to be removed. Otherwise support is a matter of mapping the watchpoint address and range the processor implementation. The DPEH watchpoint event interface will need to hook the watchpoint interrupt handler. It is likely that a generalised debug register allocation scheme will be needed along with adjustments to context switching to ensure registers used for watchpoints are global to all contexts and can easily co-exist with other uses of watchpoints within the system[12].

Porting to other operating systems.
There are four key considerations:

Module management
DProbes requires a unique handle by which it can refer to a module while either loaded or on disk. There needs to be a means of correlating a virtual storage address in a given context with a module handle. Under Linux the **inode** serves this purpose.

Page management
Probepoints need to be re-inserted when a module page is brought into memory. Under Linux DProbes achieves this by hooking the **readpage** address. If the paging mechanism is not used make the initial load of a module, as in the case of Linux kernel modules then module load and unload will also need to be hooked.

Symbolic support
To support symbolic expressions the expression-analyser in the dprobes command will need to be adapted to process the module format. Under Linux DProbes assumes the ELF format, which is common to many UNIX-like platforms.

Memory management services
Apart from basic allocation and de-allocation functions, DProbes will require a means of aliasing a physical page with a private writeable virtual address to be able to store the breakpoint instructions without causing a fault or a proliferation of privatised pages, which would be the case where a Copy-on-Write page management scheme is implemented.

Fault handling
The DPEH needs to intercept faults relating to access violations before any operating system processing so that they may be *silently* handled by the DPEH RPN command interpreter.

9. Where to obtain DProbes and GKHI:

DProbes, and GKHI are available from the IBM Linux Technology Centre's web page at:
http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes

The development team comprises:
Richard J Moore (DProbes Project Lead) - richardj_moore@uk.ibm.com
Bharata B Rao - rbharata@in.ibm.com
Subodh Soni - ssubodh@in.ibm.com
Vamsikrishna Sangavarapu - r1vamsi@in.ibm.com
Suparna Bhattacharya - bsuparna@in.ibm.com

10. References

<1> Dynamic Probes is an open-source project distributed freely under the GNU GPL from **http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes**
<2> Dynamic Probes and Generalised Kernel Hooks paper published in the USENIX Proceedings of the October 2000 Annual Linux Showcase.
<3> The SGI [**] Kernel Debugger is an open-source project from Silicon Graphics Inc.. It may be obtained from: **http://oss.sgi.com/projects/kdb**
<4> The SGI Kernel Crash Dump is an open-source project from Silicon Graphics Inc.. It may be obtained from: **http://oss.sgi.com/projects/lkcd**
<5> The Linux Trace Toolkit is an open-source project from Opersys, Motreal. It may be obtained from: **http://www.opersys.com/LTT/**
<6> Generalised Kernel Hooks Interface is an open-source project distributed freely under the GNU GPL from: **http://oss.software.ibm.com/developerworks/opensource/linux/projects/dprobes**
<7> OS/2 Trace facilities are described in the OS/2 Debugging Handbook Volume 3. Order number SBOF 8617 or as an on-line Redbook under order number SG244640.
<8> IA32 and IA64 are abbreviations for the 32-bit Pentium and 64-bit Itanium processors of the Intel Corporation [**].

11. Trademarks

[*] IBM, OS/2, zSeries, S/390 and Presentation Manager are trademarks of the International Business Machines Corporation in the United States and other countries.
[**] UNIX is a registered trademark of The Open Group in the United States and other countries.

12. Notes

[1] An RPN language is used for the following reasons:
  a. it allows a simple abstraction of the processor architecture to be defined to give access to the lowest level resources for minimal overhead.
  b. it provides a basis on which high-level language interfaces can be defined and be largely architecturally independent. Compare this with the Java [**] language and its implementation by a Java Virtual Machine which has an RPN-based virtual machine code.

[2] An interactive interface could always be provided by transfering control to a debugger such as the SGI kernel debugger.

[3] The reasons for this restrictive behavior are described in <2>. In summary this is due to the fact the recursion cannot be tolerated by the DPEH since few system services are available to it, in particular memory allocation. It would be possible to tolerate a finite level of recursion using a DPEH state saving stack independent of the IA32 implemented stack, however, performance and boundary conditions become complications. The latter in particular, since it would be difficult to manifest a consistent behavior to the user.

[4] These experiments were subsequently repeated using an Intel 200MHz Pentium processor. The results were consistent with those obtained earlier using the Intel 90Mhz Pentium processor, being scaled by a factor of approximately 50%.

[5] The **abort** RPN instruction causes probe handler to exit without calling any external logging function.

[6] **push eax** stores the value of the EAX register on the RPN stack. The processing by the interpreter for this instruction similar to that of most of the RPN instruction set.

[7] The **exit** RPN instruction causes the probe handler to exit and for the default external logging function to be called.

[8] A tracepoint is a probepoint used for the purpose of tracing.

[9] Static as opposed to dynamic trace refers here to tracepoints that are hard coded in program source as opposed to dynamically inserted at run-rime. With static trace there is always an overhead even when the tracepoint is inactive.

[10] Watchpoints refer to processor implemented breakpoints that require no code modification. In general they are implemented using special registers and features of the processor. They normally are not confined to monitoring execution but also permit memory references to be monitored. Watchpoints are usually global in nature being specified by virtual or even physical address location under some architectures.

[11] The IBM Dprobes team is working on a current project to implement a high-level language preprocessor for DProbes which generates RPN instructions from a c-like probe definition language.

[12] The IBM DProbes team submitted a Linux kernel patch to the Linux Kernel Mailing List to achieve this for Linux under IA32.