Open Access

Efficient Conditional Tracepoints in Kernel Space

Rafik Fahem and Michel Dagenais*

Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, P.O. Box 6079, Succ. Downtown, Montreal, Quebec, H3C 3A7, Canada

Abstract: With kernel static tracepoints, it is now possible to add instrumentation to the Linux kernel and obtain a low overhead trace of the whole system. However, these static tracepoints may be insufficient to diagnose the source of a functional or performance problem. Dynamic instrumentation fills the gap by enabling the insertion of additional tracepoints in other locations at run time.

This article presents a new approach for tracing the Linux kernel with dynamic and static tracepoints. These tracepoints will be conditional. Conditions are defined using complex expressions that employ the code variables and make use of arithmetic and logic operations. These expressions are written using C-like syntax.

Both static and dynamic tracepoints will evaluate and collect expressions similar to those used for conditions. In addition, static tracepoints will collect the static tracepoint data, as defined by the TRACE_EVENT macro used to define tracepoints in the Linux kernel.

Our tool was implemented based on GDB and KGTP, which is a GDB stub in kernel-space that partially implements dynamic tracepoints.

Keywords: Dynamic tracing, linux kernel, GDB, TRACE_EVENT, Systemtap, Ftrace, LTTng.

1. INTRODUCTION

With kernel static tracepoints [1, 2] defined using TRACE_EVENT [3] and user-space tracepoints provided by the UST [4] library, it is now possible to add instrumentation and obtain a low overhead trace of the whole system. However, these static tracepoints may be insufficient to diagnose the source of a problem. Dynamic instrumentation fills the gap by enabling the insertion of additional tracepoints in other locations at run time.

Recently, GDB [5] was enhanced to support dynamic tracepoints in user-space. Using this feature, tracepoints can be defined in almost every location in a program. A set of actions can be associated to each tracepoint. These actions may be used to collect the values of the registers at the time the tracepoint was hit or to evaluate user-defined expressions. These expressions may be complex and can reference all the program variables accessible from the tracepoint location. GDB being able to read the program debug information and to locate variables, we can refer to variables in these expressions by their name without having to care about their location. GDB static and dynamic tracepoints may be conditional. In this case, expressions can be used as conditions. In order to simplify evaluation, GDB converts expressions used in conditions and actions to bytecode [6, 7] which is interpreted each time the corresponding tracepoint is hit. Moreover, in some situations, GDB converts the conditions' bytecode into native code in order to improve performance.

More recently, the KGTP (Kernel GDB Tracepoints)[8] kernel module was submitted as a contribution to the Linux kernel. It uses kprobes [9-11] to insert GDB dynamic tracepoints into the kernel, implements the RSP (Remote Serial Protocol) to communicate with GDB and can interpret the bytecode used by GDB to define conditions and actions. However, KGTP is unable to convert this bytecode to native code.

The goal of this work was to extend the KGTP module by implementing a bytecode to native code converter in kernel space for both conditions and actions. GDB was also integrated with TRACE_EVENT through KGTP in order to be able to list, enable and disable the kernel static tracepoints. Expressions may be used in conditions and additional actions. These expressions are converted to native code and may reference any variable accessible from the static tracepoint location.

2. PREVIOUS WORK

Dtrace [12, 13] is a tracing tool developed for the Solaris kernel and ported to other platforms such as Mac OS, QNX and FreeBSB. Static tracepoints can be inserted in the kernel source code using a C macro which expands to a non-existing function call. This function call is replaced by a NOP operation at link-time. The linker saves the function name and the call address. In order to enable the tracepoint, DTrace uses this saved information to replace the NOP by a probe which has the same name as the non-existing function.

*Address correspondence to this author at the Department of Computer and Software Engineering, Ecole Polytechnique de Montreal, P.O. Box 6079, Succ. Downtown, Montreal, Quebec, H3C 3A7, Canada;
Tel: 1 514 340 4711; E-mail: michel.dagenais@polymtl.ca

Dynamic tracepoints [14] are used to avoid the overhead caused by disabled static tracepoints. They can only be inserted in the kernel functions entry and exit points. They are implemented on the x86 architecture using a trap [15]. When the tracepoint is hit, the instruction transfers control to DTrace.

Static and dynamic tracepoints [16] in DTrace are defined using D scripts. D is a language which has a C-like syntax. Conditions can be associated to both tracepoints. D scripts are converted to an intermediate code. The intermediate language is a RISC instruction set. This code is emulated each time the tracepoint is hit.

Several efforts were made in the Linux community to bring an answer to Dtrace. Some of them are presented in the subsequent paragraphs.

Systemtap [17-19] is a tracing tool similar to DTrace which implements dynamic and static tracepoints in the Linux kernel. Systemtap uses scripts in order to define both tracepoints. These scripts are compiled into kernel modules before tracing starts.

Dynamic tracepoints are implemented in Systemtap using kprobes. They can be conditional. Conditions are defined using C-like complex expressions. All the variables accessible from the dynamic tracepoint address can be used in these expressions. Dynamic tracepoints are able to evaluate and collect expressions that have the same characteristics as the expressions used for conditions.

Systemtap is also able to connect to the kernel static tracepoints defined using TRACE_EVENT. Conditional expressions are limited to using the variables passed to the Systemtap registered probe, as defined in the TRACE_EVENT macro. Static tracing capabilities are therefore limited. In addition, Systemtap is unable to list the probe points, which makes it require a certain level of familiarity with the Linux kernel source code.

Ftrace [20, 21] is another Linux kernel tracing tool that is part of the mainline kernel. It offers both dynamic and static tracepoints. Dynamic tracepoints [22, 23] are based on kprobes. Because Ftrace is unable to read the kernel debug information, we have to manually specify the exact address location of each variable to collect.

Ftrace connects to the kernel static tracepoints defined with TRACE_EVENT. These tracepoints are only able to collect the data defined in the TRACE_EVENT declaration.

Ftrace is unable to associate conditions to both tracepoints. Only filters can be used. These filters have limited capabilities compared to SystamTap and GDB conditional expressions.

In conclusion, we notice that each tracing tool in the Linux kernel lacks some functionality. Some tools, such as Ftrace or LTTng offer great performance at the expense of flexibility. Indeed, Ftrace is unable to insert conditional tracepoints in the kernel.

On the other side, Systemtap offers such functionality. It is able to insert conditional dynamic and static tracepoints based on the kernel debugging information. However, the integration between Systemtap and TRACE_EVENT is not optimal and a complicated setup is required on the tracing target with a compiler and compilation server.

In the subsequent sections, we describe the architecture of the proposed solution. We explain the implementation of the bytecode to native code translator used in both dynamic and static tracepoints. We then describe the method used to integrate KGTP with the kernel static tracepoints and the modifications we had to apply to the existing TRACE_EVENT implementation in order to collect the registers needed to evaluate expressions. Finally, we will discuss how to integrate these tracepoints with LTTng in order to increase performance.

3. METHODOLOGY

3.1. Bytecode to Native Code Translation

Converting the bytecode produced by GDB to native code has proved its efficiency in user-space especially for conditional tracepoints. Table **1** shows the execution times that we collected during the experiments. In order to minimize the overhead of executing dynamic and static tracepoint probes in kernel space, we implemented the translator in KGTP.

Table 1. Bytecode *vs* Native Code in User-Space

Processing	Bytecode	Native
Condition	444 (ns)	115 (ns)
Data	3.075 (µs)	2.9 (µs)

Similarly to what GDBServer does in user-space, KGTP translates the bytecode using a one-to-one translation scheme, meaning that for each agent expressions opcode, there is a corresponding assembly code snippet.

Fig. (**1**) shows the assembly code corresponding to the “add” opcode.

As Fig. (**1**) shows, each native code snippet is located in a separate function and is produced using inline assembly. The assembly code is located between two labels preceded by a jump to the second label. The jump instruction is useful to avoid executing that code when the function is called and the two labels are needed to get the start and end addresses of the native code in the program memory space. Having these two addresses, we just have to run what is in between to execute an “add” instruction.

In some situations, inline assembly is not sufficient to produce the correct native code. In fact, some opcodes have arguments with values changing from an expression to another. For example, the “const8” opcode pushes an eight-bit constant on the stack. This constant is provided in the bytecode and cannot be guessed when compiling the KGTP module. Thus, we have to overwrite the native code produced in order to put the right value. Fig. (**2**) illustrates this case.

```
#define EMIT_ASM(BUFFER,INDEX,NAME, INSNS)                       \
  do                                         \
      [                                       \
      extern unsigned char start_ ## NAME, end_ ## NAME;        \
      __asm__ ("jmp end_" #NAME "\n"                        \
           "\t" "start_" #NAME ":"                          \
           "\t" INSNS "\n"                             \
           "\t" "end_" #NAME ":");                          \
    copy_into_buffer(BUFFER,INDEX,&start_ ## NAME,&end_ ## NAME);\
    } while (0)

static void emit_add(unsigned char *dest, int *offset)
 [
    EMIT_ASM (dest,offset,add,
            "add (%rsp),%r15\n\t"
            "lea 0x8(%rsp),%rsp\n\t"
            "dec %r14");
}
```

Fig. (1). "Add" assembly code.

This is also the case for the “goto” and “if_goto” opcodes.

The native code translator that we implemented works in five steps:

- It allocates a virtually contiguous, executable memory. This buffer will contain the native code produced. It also allocates two other tables. The first one is a mapping table used to map the address of each opcode in the bytecode to the address of the corresponding native code snippet in the executable buffer. The second one is used to store the addresses of the “goto” and “if_goto” instructions in the bytecode. These two tables are used to determine and update the target addresses of the jump instructions used in the “goto” and “if_goto” opcodes.
- It starts by copying the function prologue into the executable buffer.
- It iterates over the bytecode and copies the corresponding native code snippets into the buffer. Overwrites are performed when needed (const8, zero_ext...).
- It copies the function epilogue into the buffer.
- It updates the jump addresses using the two tables.

  Because the space allocated for the buffer is executable, the native code that it contains can be executed by casting the pointer to the buffer into a function pointer and calling that function.

Both dynamic and static tracepoints should be able to verify conditions and collect user defined expressions. Therefore, this translation is applied to the conditions and actions of all the tracepoints.

3.2. Listing and Enabling Static Tracepoints

The user is not supposed to know the exact location and name of every static tracepoint. We then have to be able to list the static tracepoints defined in the kernel upon request. Besides, because this information may change at any time and new static tracepoints can be added to the kernel, we cannot statically store it in the KGTP module. It must be retrieved at run time from the kernel.

Based on what is done in Ftrace, we used a static memory area in the kernel to store this information. By having the start and end addresses of this memory area, we can access it from both the kernel and the KGTP module, to read and write the information needed. We defined a structure that will contain all that data. For each static tracepoint, we have a corresponding instance of that structure that we called “kgtp_event_call” in the static memory area. Fig. (**3**) shows that structure.

The collect_regs and collect_sdata members are used to determine whether or not we have to collect the registers and the static tracepoint data before executing the KGTP probe. The *gentry* pointer is used internally in KGTP and stores, among other things, the pointers to the KGTP buffers. The *event_name* and *trace_system* contain the trace event name and system as defined in the TRACE_EVENT macro call.

```
static void emit_const8(unsigned char *dest, int *offset,LONGEST arg)
 [
    EMIT_ASM (dest,offset,const8_1,
            "movabs $0xffffffffffffffff,%r15");
    *((LONGEST*)(dest+*offset-8)) = arg;
}
```

Fig. (2). Overwriting the native code.

```
struct kgtp_event_call

  [
     long int collect_regs;
     long int collect_sdata;
     void (*probe)(struct kgtp_event_call*, struct pt_regs, char*);
     void *gentry;
     char *event_name;
     char *trace_system;
     void *address;
     int (*condition_function)(struct kgtp_event_call*, struct pt_regs);
 };
```

Fig. (3). Structure used to store the tracepoint information.

The *address* member contains the location of the static tracepoint in the kernel address space. Finally, the *condition_function* and *probe* function pointers contain the pointers to the condition function and KGTP probe respectively.

The structure data is written only at compile-time or by KGTP before starting the tracing session. Therefore, there is no data corruption risk in case we have several threads accessing that data at the same time.

The structure is created at the static tracepoint call site. Fig. (**4**) shows the code to create that structure. First of all, the tracepoint and the trace system names are stored in the corresponding sections. After that, we move to the _kgtp_event_calls section and create the kgtp_event_call structure. At that time, we can only write the tracepoint and trace system name, and the tracepoint address which corresponds to the address from which the tracepoint condition is verified.

As shown in Fig. (**4**), the kgtp_event_call structures are created in a section called _kgtp_event_calls. Knowing the size of the structure and the start and end addresses of that section, we can iterate to list the static tracepoints. In order to

```
asm volatile( \
    ".ifndef __mstrtab_" __stringify(name) "\n\t" \
    ".section ___kgtp_event_calls_strings1,\"aw\",@progbits\n\t" \
    "__mstrtab_" __stringify(name) ":\n\t" \
    ".string \"" __stringify(name) "\"\n\t" \
    ".previous\n\t" \
    ".endif\n\t" \
    ); \
\
asm volatile( \
    ".ifndef __mstrtab_" __stringify(TRACE_SYSTEM) "\n\t" \
    ".section ___kgtp_event_calls_strings2,\"aw\",@progbits\n\t" \
    "__mstrtab_" __stringify(TRACE_SYSTEM) ":\n\t" \
    ".string \"" __stringify(TRACE_SYSTEM) "\"\n\t" \
    ".previous\n\t" \
    ".endif\n\t" \
    ); \
\
asm volatile( \
    ".section _kgtp_event_calls,\"aw\",@progbits\n\t" \
    "1:\n\t" \
    _ASM_PTR "0\n\t" \
    _ASM_PTR "0\n\t" \
    "19:\n\t" \
    _ASM_PTR "0\n\t" \
    _ASM_PTR "0\n\t" \
    _ASM_PTR "(__mstrtab_" __stringify(name) ")\n\t" \
    _ASM_PTR "(__mstrtab_" __stringify(TRACE_SYSTEM) ")\n\t" \
    _ASM_PTR "(18f)\n\t" \
    _ASM_PTR "0\n\t" \
    ".previous\n\t" \
```

Fig. (4). Creating the kgtp_event_call structure.

do that, we had to modify the system image layout by adding the _kgtp_event_calls section to the linker script. We are then able to get the start and end addresses by calling the get_start_kgtp_event_calls and get_stop_kgtp_event_calls functions. Fig. (**5**) shows the modifications brought to the include/asm-generic/vmlinux.lds.h file.

Listing the static tracepoints is done in GDB using the “info static-tracepoint-markers”. Because GDB has no direct access to the tracepoint data in the static memory area, it communicates with KGTP using the appropriate requests to get that information. The two requests used in this case are “qTfSTM” and “qTsSTM”. The first request asks the remote stub, which is KGTP in our case, to return the information about the first static tracepoint. If the response that we return to GDB is successful, then GDB keeps sending “qTsSTM” requests and waiting for the response in order to get the information about the next static tracepoint. This process stops when we get to the end of the static memory area. In that case, we return an empty message to GDB.

3.3. Collecting the Registers

In addition to the case where the user asks GDB to collect the registers using the “collect $regs” command, GDB may need the values of the registers at the moment the tracepoint was hit, in order to evaluate a condition or to evaluate an expression. GDB is able to find which register is used to store the value of a certain variable at that exact moment. Therefore, we have to collect the registers before calling the KGTP probe.

In order to avoid compiler inserted code which may modify the values of general purpose registers, the tracepoint address that is returned to GDB corresponds to the instruction that follows the code that does the collection. That way, we are always sure that the values recorded from the registers correspond to what GDB is asking for. Register collection is done using extended inline assembly. A pt_regs structure is provided as an input operand. That structure is created on the stack, and not in the static memory area corresponding to that tracepoint, in order to avoid memory corruption issues in case it is hit by multiple threads. Because we need at least one register to store the address of the pt_regs structure, we had to push the RAX register on the stack. After moving the reference to the pt_regs structure to that register, we pop the stored value directly to its corresponding location in the structure. We then copy the

```
#define KGTP_EVENT_CALLS()
VMLINUX_SYMBOL(__start_kgtp_event_calls)=.;\
    *(_kgtp_event_calls)                \
    VMLINUX_SYMBOL(__stop_kgtp_event_calls) = .;

#define DATA_DATA                           \
    *(.data)                            \
    *(.ref.data)                            \
    *(.data..shared_aligned) /* percpu related */           \
    DEV_KEEP(init.data)                     \
    DEV_KEEP(exit.data)                     \
    CPU_KEEP(init.data)                     \
    CPU_KEEP(exit.data)                     \
    MEM_KEEP(init.data)                     \
    MEM_KEEP(exit.data)                     \
    . = ALIGN(32);                          \
    VMLINUX_SYMBOL(__start___tracepoints) = .;          \
    *(__tracepoints)                        \
    VMLINUX_SYMBOL(__stop___tracepoints) = .;           \
    /* implement dynamic printk debug */            \
    . = ALIGN(8);                           \
    VMLINUX_SYMBOL(__start___verbose) = .; \
    *(__verbose) \
    VMLINUX_SYMBOL(__stop___verbose) = .;           \
    LIKELY_PROFILE()                            \
    BRANCH_PROFILE()                        \
    TRACE_PRINTKS()                         \
                                    \
    STRUCT_ALIGN();                         \
    FTRACE_EVENTS()                         \
    KGTP_EVENT_CALLS();                     \
                                    \
    STRUCT_ALIGN();                         \
    TRACE_SYSCALLS()
```

Fig. (5). Modifications brought to the linker script.

rest of the registers.

Collecting the registers is not always necessary. In fact, if the tracepoint is disabled or no tracepoint condition is specified, and only the tracepoint static data is collected, the registers collected will not be useful. We thus check if the tracepoint is enabled and KGTP needs the registers before collecting them. This is done using the “probe” and “collect_regs” fields in the kgtp_event_call structure. If the “probe” function pointer is NULL, this means that the KGTP probe is not registered and therefore, the tracepoint is disabled.

Because the assembly “TEST” instruction accepts only register operands, we used the same RAX register. After saving its value on the stack, we load the field in the register and call the test instruction. If saving the registers is needed, we then execute the code described in the previous paragraph. Otherwise, we directly call the KGTP probe without even restoring the RAX register. In fact, this register is listed as a clobbered register and therefore, the compiler will consider these changed. The same technique is used to test if the tracepoint is enabled.

3.4. Extracting the TRACE_EVENT Data

Each TRACE_EVENT defines the parameters passed to the registered probes. The number and the types of these parameters vary from a tracepoint to another. Using a single probe for all tracepoints is therefore unfeasible.

```
#undef __array
#define __array(type, item, len)    type    item[len];

#undef __field
#define __field(type, item)     type    item;

#undef __field_ext
#define __field_ext(type, item, filter_type)    type    item;

#undef __dynamic_array
#define __dynamic_array(type, item, len) \
    type* item;\
    int item##_len;

#undef __string
#define __string(item, src) char* item;

#undef TP_STRUCT__entry
#define TP_STRUCT__entry(args...)   args

#undef TP_PROTO
#define TP_PROTO(args...)

#undef TP_ARGS
#define TP_ARGS(args...)

#undef TP_fast_assign
#define TP_fast_assign(args...)

#undef TP_printk
#define TP_printk(args...)

#undef TP_perf_assign
#define TP_perf_assign(args...)

#undef TRACE_EVENT
#define TRACE_EVENT(name, proto, args, tstruct, assign, print) \
    struct kgtp_event_##name##__entry [ \
        tstruct \
    };

#undef DECLARE_EVENT_CLASS
#define DECLARE_EVENT_CLASS(name, proto, args, tstruct, assign, print)\
    struct kgtp_event_##name##__entry [ \
        tstruct \
    };

#undef DEFINE_EVENT
#define DEFINE_EVENT(template, name, proto, args)

#include TRACE_INCLUDE(TRACE_INCLUDE_FILE)
```

Fig. (6). Creating the __entry structure.

Therefore, each static tracepoint needs its own function that will accept the parameters and extract the appropriate fields. Writing such a function for each TRACE_EVENT manually can solve the problem but, in that case, KGTP won't be able to connect to the new static tracepoints added to the kernel, and we will find ourselves writing a new function each time we define a TRACE_EVENT. SystemTap suffers from this limitation. That function will extract the fields defined in the TRACE_EVENT from the parameters passed to it.

Based on the integration between TRACE_EVENT and Ftrace, we defined the functions to connect to the static tracepoints and the intermediate data we need using two stages. Figs. (**6**, **7**) illustrate this.

In the first stage, we define the __entry structure as declared in the TRACE_EVENT. This structure contains all the fields that will be extracted from the parameters passed to the probe. As Fig. (**6**) shows, because we are only defining the structure here, we had to consider only the “name” and “tstruct” parameters of the TRACE_EVENT macro. The other tokens are simply ignored.

The first token is used as part of the structure name. The second one declares the fields inside the structure. By putting the tstruct token between the brackets([}), the preprocessor will define the structure fields using the corresponding macros (__array, __field, __dynamic_array... etc).

```
#undef __dynamic_array
#define __dynamic_array(type, item, len) \
    __entry->item##_len = len;

#undef __string
#define __string(item, src)

#undef __array
#define __array(type, item, len)

#undef __field
#define __field(type, item)

#undef __field_ext
#define __field_ext(type, item, filter_type)

#undef __assign_str
#define __assign_str(dst, src) __entry->dst = (char*)src;

#undef tp_assign
#define tp_assign(dest, src)        __entry->dest = src;

#undef tp_memcpy
#define tp_memcpy(dest, src, len)   memcpy(__entry->dest, src, len);

#undef tp_memcpy_dyn
#define tp_memcpy_dyn(dest, src, len) __entry->dest = src;

#undef tp_strcpy
#define tp_strcpy(dest, src)        __assign_str(dest, src);

#undef TP_fast_assign
#define TP_fast_assign(args...) args

#undef TP_PROTO
#define TP_PROTO(args...) args

#undef TP_ARGS
#define TP_ARGS(args...) args

#undef TRACE_EVENT
#define TRACE_EVENT(name, proto, args, tstruct, assign, print) \
  void get_##name##_kgtp_string(char* buffer, proto) [ \
   struct kgtp_event_##name##__entry global__entry_##name; \
   struct kgtp_event_##name##__entry *__entry = &global__entry_##name;\
        struct trace_seq __maybe_unused *p = &kgtp_seq_struct; \
        tstruct; \
        assign; \
        snprintf(buffer,500,print); \
    } \
    EXPORT_SYMBOL(get_##name##_kgtp_string);
```

Fig. (7). Function to extract the tracepoint data.

In the second stage, we create our function. The “name” token is pasted to the function name. The “proto” token is used to declare the function arguments list. The “tstruct” token is used to save dynamic arrays lengths if any. The “assign” token is used to generate the code to fill in the __entry structure. All the fields in the structure are a copy of the original parameters, except for dynamic arrays and strings. For these two cases, we wanted to avoid dynamically allocating memory, especially as these fields will be used only to generate the string and will no longer be needed afterwards. That is why we only copy the pointers to dynamic arrays and strings and use the original data without copying it. Finally, the “print” token is used to define the format string and to pass the appropriate arguments to the snprintf function that generates the function and copies it to a buffer.

Basically, the function creates the structure corresponding to that TRACE_EVENT, fills it using the parameters passed, generates the string using that structure and finally copies it to the buffer specified as a parameter to the function.

3.5. Condition Evaluation and Data Collection

Recall that generating the string from the tracepoint parameters can only be done in the tracepoint site and cannot be moved to the KGTP module. In a previous implementation, we called the KGTP probe and passed the string we generated to it as a parameter. Because the condition was verified inside that probe, we found ourselves extracting the data for nothing in the case the condition was false. That is why we split the KGTP probe and implemented it in two functions.

The first function takes the pt_regs structure containing the saved registers and verifies if the condition is true. Then, depending on the result returned, we generate the TRACE_EVENT data and finally call the second KGTP function. Once again, in order to be thread safe, that data is passed to the function on the stack and not in the static memory area. Fig. (**8**) shows the stack when this function is called.

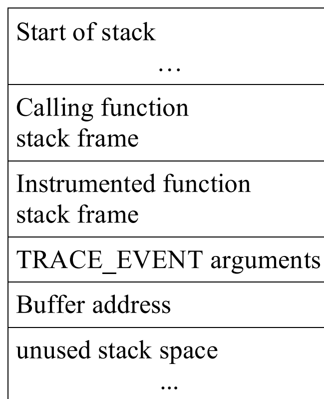

Fig. (8). Calling the first function.

The second function executes the actions defined by the user one by one, similarly to what is done with dynamic tracepoints. In the case of the “collect $_sdata” action that collects the TRACE_EVENT data, KGTP copies the string collected on the tracepoint site to the buffers. First of all, KGTP needs to insert a frame head in that space to be able to identify that data when reading it afterwards. After that, it copies the string collected preceded by its length. Fig. (**9**) shows the stack when the KGTP probe is called.

3.6. Algorithm Description

In the previous sub-sections, we described the different parts of the algorithm used to collect TRACE_EVENTs data and to evaluate expressions in KGTP. In this sub-section, we will put these pieces together. The following pseudocode presents the modifications we had to apply to the static tracepoints sites in order to integrate KGTP with the Linux kernel:

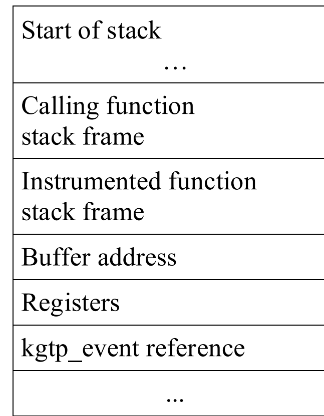

Fig. (9). Calling the second function.

```
create the kgtp_event_call structure
   if (tracepoint_enabled)
      if (need_to_collect_the_registers)
         save the registers in the pt_regs structure
     if (no_condition OR condition_is_true)
        if (need_to_collect_TRACE_EVENT_data)
           call function to extract the data
        call the KGTP probe
call the old trace function
```

Other tracing tools like Ftrace are able to connect to static tracepoints using the functions provided by TRACE_EVENTs. Our situation was a bit more complicated. In fact, we needed to save the registers in order to be able to verify conditions and to evaluate expressions. These features are not implemented in Ftrace. We could try to recover the registers from the call stack, but we have to note that calling the function that extracts the data from the tracepoint parameters cannot be called from inside the probe, and thus we cannot avoid altering the code in the tracepoint site.

4. RESULTS

The following section shows the results of running KGTP in both dynamic and static tracing modes. These results are compared with those obtained with Systemtap. Finally, we discuss the performance of combining KGTP

and LTTng [24, 25] in static tracing mode with the performance of Systemtap.

In order to make sure the results are repeatable and accurate, we inserted dynamic and static tracepoints in a dummy function that does nothing but incrementing a counter. The function, called kgtp_test_function, was defined in kernel/module.c. The benchmark was executed with KGTP and GDB running on the same machine. This machine contains an Intel Xeon E5405 (4 cores, 4 threads) at 2GHz and 8GB of memory.

The probes execution times were measured in cycles, with a kernel module that calls the test function 10.000 times. Fig. (**10**) shows the test function and Fig. (**11**) shows the loop used.

4.1. Dynamic Tracepoints with Native Code Support

The goal of this test case is to compare the performance increase obtained by converting the bytecode produced by GDB into native code in dynamic tracing mode. The results are then compared to those produced by SystemTap, where the probes are converted to C code and compiled by GCC.

The test probe was inserted at the address of the instruction following the test function prologue for both KGTP and Systemtap. We first measured the execution time of the condition alone. In order to do that, we had to make sure the condition was always false. The second step was to measure the time needed to evaluate and store an expression. The last one was a combination of a true condition and an expression. The expressions used employ the two parameters of the dummy function (counter1 and counter2).

Fig. (**12**) shows the GDB commands used to configure tracing and to output the trace for the third test case.

The two tools use Kprobes to connect dynamic tracepoints to the kernel. The tracepoint address was chosen so that Kprobes was able to optimize the probe and use a jump instead of the int3 interrupt. The execution times presented in the table below were calculated from the moment the jump instruction was met until the execution of the original instruction. This was done by calculating the difference between the dummy function execution time with and without a kprobe connected. Thus, the numbers presented below include the overhead added by kprobes.

Table 2. Execution Times for Dynamic Tracepoints

Computation	KGTP with Native Code (Cycles)	Systemtap (Cycles)
False condition: 2xarg1+3xarg2<0	202	351
Expression only: 2xarg1+3xarg2	500	1035
Condition + expression: 2xarg1+3xarg2>0	602	1061

Table **2** shows that KGTP is always faster than Systemtap, whether to evaluate conditions or to store the expressions.

Based on these results, we may think that executing the native code produced by KGTP from the bytecode is faster than the optimized native code produced from the Systemtap script by the compiler. After analyzing the temporary C files produced by Systemtap, we discovered that the tool adds some “setup code” that is always executed at the start of the probe.

For KGTP, the execution time in the third test case is nearly the sum of the execution times of the first two test cases, if we substract the time taken to setup the kprobe(115 cycles for an optimized kprobe). That is not true in the case of Systemtap because of the setup code that is always executed.

Assuming that evaluating the conditions in the first and third cases takes nearly the same time, we can conclude that executing the native code produced by our implementation takes 102 cycles for the two expressions. For Systemtap, the second and third cases let us conclude that the native code produced takes 26 cycles to be executed. This big difference may be explained by two reasons.

```
int kgtp_counter=1;
int kgtp_test_function(int counter1, int counter2)
 [
    kgtp_counter++;
    __trace(kgtp_module_event,counter1,counter2);
}
```

Fig. (10). Test function.

```
set_current_state(TASK_INTERRUPTIBLE);

time1 = get_timestamp();
for (i = 0; i < NR_LOOPS; i++)  [
    counter1++;
    kgtp_test_function(counter1, counter2);
    counter2++;
}
```

Fig. (11). Loop used to calculate the execution times.

```
root@rafik-desktop:/usr/src/linux-2.6.37# gdb vmlinux
(gdb) target remote /proc/gtp
Remote debugging using /proc/gtp
0x0000000000000000 in ?? ()
(gdb) trace *0xffffffff810a3c50 if (2*counter1+3*counter2>0)
Tracepoint 1 at 0xffffffff810a3c50: file kernel/module.c, line 118.
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
>collect (2*counter1+3*counter2)
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind start
Found trace frame 0, tracepoint 1
#0 kgtp_test_function (counter1=1, counter2=0) at kernel/module.c:118
118    [
(gdb) tdump
Data collected at tracepoint 1, trace frame 0:
(2*counter11+3*counter2) = 2
(gdb) tfind
Found trace frame 1, tracepoint 2
118
(gdb) tdump
Data collected at tracepoint 1, trace frame 1:
(2*counter1+3*counter2) = 7
```

Fig. (12). GDB dynamic tracepoints.

The first one is that the native code produced by KGTP is put in a function. Thus, we cannot neglect the execution time of the function prologue and epilogue. In the case of Systemtap, the condition evaluation and the data collection are done directly in the body of the kprobe handler.

The second one is that the native code produced by KGTP is a strict translation of the bytecode that GDB generated. Thus, its performance depends on how optimized the bytecode is. For example, the condition used in the third test (2*param1+3*param2>0) is translated by GDB into the bytecode shown in Table **3**.

If we look at the five "ext" instructions, they all pop the value that was pushed in the previous instruction. Thus, we have five successive pop/push operations that could be avoided. In addition, the three "ext" instructions that follow the "const8" instruction are unnecessary because the constants that we pushed are already sign-extended to zero. We can then conclude that the bytecode produced by GDB could be optimized, improving the native code significantly. Intermediate code optimization is not a new concept. It is used for example in Valgrind to make the intermediate representation of the original binary code more efficient.

Finally, we have to note the impact of the bad performance of KGTP buffers. In fact, the expression collected in the second test case is very similar to the condition evaluated in the first case. The main difference between their bytecodes is that the expression collected contains additional trace opcodes. These opcodes trace the values of the variables used in the expression. We notice that these opcodes are the main cause of the significant difference between these two cases. Thus, by using more efficient data structures, like ring buffers to store the trace, the time taken to execute the "trace" and "trace_quick" opcode can be reduced. Moreover, by integrating KGTP with other tracing tools like LTTng, we can make use of their fast tracing capabilities. Storing the trace would then be performed by LTTng.

4.2. Static Tracepoints

The static tracepoint was created by defining a new TRACE_EVENT, as shown in Fig. (**13**). The expression defined in this static tracepoint is very similar to the one defined in the dynamic tracepoint test. It collects the two parameters passed to the dummy function.

KGTP is able to connect to the static tracepoint and collect the data as defined by the TP_printk macro

Table 3. Bytecode for the Condition Expression

Instructions	Description
0x22 0x02	const8: push the 8-bit integer 2 on the stack, without sign extension
0x26 0x00 0x05	reg: push the value of the register 5, without sign extension
0x16 0x20	ext: pop an unsigned value from the stack. All bits to the left of bit 31(where the least significant bit is bit 0) are set to the value of bit 31.
0x04	mul: pop two integers from the stack, multiply them, and push the product on the stack.
0x16 0x20	ext
0x22 0x03	const8
0x26 0x00 0x04	reg
0x16 0x20	ext
0x04	mul
0x16 0x20	ext
0x02	add: pop two integers from the stack, and push their sum, as an integer.
0x16 0x20	ext
0x22 0x00	const8
0x2B	swap: exchange the top two items on the stack.
0x14	less_unsigned: pop two signed integers from the stack. If the next-to-top value is less than the top value, push the value one. Otherwise, push the value zero.
0x27	end: stop executing bytecode.The result should be the top element of the stack.

automatically. On the other side, Systemtap is only able to trace the parameters given to the probe registered to the tracepoint automatically. In order to get the same results with the two tools, we have to redefine the way to extract the data in the Systemtap script. Figs. (**14**, **15**) show the KGTP and Systemtap scripts used to connect to the tracepoint.

As a first step we tried to measure the overhead of a disabled static tracepoint in the kernel. After that, we compared the performances of KGTP and Systemtap.

4.2.1. KGTP Kernel Overhead

In order to avoid collecting the registers when the KGTP tracepoint is disabled, we check if the KGTP probe was registered in the static memory area corresponding to the tracepoint that was hit. We wanted to measure the overhead caused by this additional code. In order to do that, we calculated the execution time of the dummy function before and after the changes were applied to the kernel. Table **4** shows the results.

Table 4. KGTP Kernel Overhead

	Function Execution Time (Cycles)
Before	12
After	19

We may conclude that a disabled static tracepoint costs an extra 7 cycles. This can be explained by the fact that we need at least one register to proceed with the check. This is why we are saving the RAX register before calling the TEST instruction.

Saving the RAX register was unavoidable. We may want to put it in the clobbered registers list to force the compiler to use other registers for the kernel variables, but if we asked GDB to collect the registers using the « collect $regs » command in the same static tracepoint, the traced value of RAX would be incorrect.

4.2.2. KGTP vs Systemtap

As for dynamic tracepoints, we calculated the results for the three test cases. Table **5** shows the results. We have to note that our proposed extended KGTP with static tracepoints also supports our proposed bytecode to native code translator.

Table 5. KGTP *vs* Systemtap

	KGTP (Cycles)	Systemtap (Cycles)
Condition	154	223
Data	1216	1252
Condition and Data	1368	1336

Table **5** shows that KGTP is faster than Systemtap in the cases where the condition is false or there is no condition. We may think that the execution time for a probe with a true condition will nearly be the sum of the two first cases. This is only true for KGTP. Indeed, as for dynamic tracepoints, the Systemtap probe always executes the « setup code ». This explains the fact that Systemtap is faster than KGTP in the third case.

```
TRACE_EVENT(kgtp_module_event,
    TP_PROTO(int counter1, int counter2),
    TP_ARGS(counter1, counter2),
    TP_STRUCT__entry(
        __field(    int,    counter1    )
        __field(    int,    counter2    )
    ),
```

Fig. (13). TRACE_EVENT used for the test.

```
(gdb) target remote /proc/gtp
Remote debugging using /proc/gtp
0x0000000000000000 in ?? ()
(gdb) info static-tracepoint-markers
Cnt ID Enb   Address What
1 module::kgtp_module_event n      0xffffffff810a3cef in kgtp_
test_function at kernel/module.c:120 Data: ""
2 module::module_load n 0xffffffff810a6b33 in load_
module at jernel/module.c:2691     Data: ""
3 skb:net_dev_xmit n 0xffffffff814be486 in dev_h
ard_start_xmit at net/core/dev.c:2069 Data: ""
4 skb::net_dev_xmit n 0xffffffff814be72c in dev_h
ard_start_xmit at net/core/dev.c:2046 Data: ""
(gdb) strace *0xffffffff810a3cef if (2*counter1+3*counter2>0)
Probed static tracepoint marker "module::kgtp_module_event"
Static tracepoint marker 1 at 0xffffffff810a3cef: file kernel/module.c, line 120.
(gdb) actions
Enter action for tracepoint 1, one per line.
End with a line saying just "end".
>collect (2*counter1+3*counter2)
>end
(gdb) tstart
(gdb) tstop
(gdb) tfind start
Found trace frame 0, tracepoint 1
#0 0xffffffff810a3cef in kgtp_test_function (counter1=1, counter2=0)
      at kernel/module.c:120
120                 __trace(kgtp_module_event,counter1,counter2);
(gdb) tdump
Data collected at tracepoint 1, trace frame 0:
(2*counter1+3*counter2) = 2
(gdb) tfind
Found trace frame 1, tracepoint 1
0xffffffff810a3cef          120           __trace(kgtp_module_event,counter1,counter2);
(gdb) tdump
Data collected at tracepoint 1, trace frame1:
(2*counter1+3*counter2) = 7
```

Fig. (14). GDB static tracepoints.

We also notice that the two tools are slow to extract and save the data. In the case of KGTP, this is caused by how the data is generated and stored in the buffers. Unlike more optimized tools like Ftrace or LTTng, KGTP fills the __entry structure and then pretty prints it to generate a string, as defined in the TP_printk macro.

Once the string is generated, it is copied in the KGTP buffers after the appropriate space is allocated. It is clear that KGTP is not well suited for high performance tracing and

```
probe kernel.trace("kgtp_module_evnet")
 [
    if(2*$counter1+3*$counter2>0)
        printf ("time=%d count=%d\n",$counter1,$counter2)
}
```

Fig. (15). Systemtap script.

that storing the binary __entry structure instead of the string, and using more efficient data structures to record the trace, will improve the performance of the tool.

We ran a similar benchmark for Ftrace to show the difference of performance between storing strings in simple buffers and storing binary data in more efficient ring buffers. We used the same test module to run three test cases on our static tracepoint. In the first one, only the data was recorded. In the second and third ones, we associated a filter with the tracepoint. We could not use the same expression as with KGTP and Systemtap, because arithmetic operators are not suppported by Ftrace. We used a simpler expression instead. Table **6** shows the results.

Table 6. Ftrace Execution Time

	Execution Time (Cycles)
Data only	297
False filter	360
True filter	370

Table **6** shows that Ftrace is nearly four times faster than KGTP when collecting and storing the trace data in the ring buffer. This proves that the current implementation of KGTP is not optimized.

Moreover, static tracepoint conditions suffer from the same optimization problems, as discussed in the dynamic tracepoints section. By implementing the native code optimizer, we shall have faster execution times for expressions in static tracing mode.

Finally, we have to note that KGTP has another advantage compared to all the other tacing tools. In fact, thanks to the changes we applied to the kernel in order to collect the registers at the tracepoint site, and because GDB is able to read the debugging information generated at compile-time, static tracepoint conditions may use all the global and local variables accessible from the tracepoint address. Moreover, in addition to the static tracepoint data defined in the TRACE_EVENT call, GDB static tracepoints are able to execute other actions like collecting the registers and evaluating user defined expressions, as for dynamic tracepoints. Other tracing tools have limited capabilities compared to KGTP. Systemtap is limited to using the parameters passed to the registered function in the condition and in the expressions to collect. Ftrace and LTTng do not even implement conditions and are only able to collect the static tracepoint string.

4.2.3. KGTP-LTTng Integration

Currently, the time taken to have the static tracepoint data written in the KGTP buffers, from the moment the tracepoint is hit, can be calculated using the following equation:

T = Treg + Tcond + Tgen+ Tbuf

Treg is the time needed to verify whether the static tracepoint is enabled and to collect the registers if needed.

Tcond is the time taken to check if a condition is associated to the tracepoint and to evaluate it.

Tgen is the execution time of the get_##name##_kgtp_string function. Finally, Tbuf is the time needed to store the string in the KGTP buffers.

Knowing that the lack of performance of our implementation is primarily caused by how we are generating and storing data, we thought about combining the flexibility of GDB agent expressions and the high performance of LTTng. Instead of generating strings, LTTng is able to store the __entry structure into its ring buffers directly. Similarly to the way we defined the function that generates the string from the TRACE_EVENT, using macro redefinitions, LTTng is able to store the __entry structure metadata for every static tracepoint. It is thus able to extract the appropriate data from the structure and produce the pretty printed string when the user is reading the trace.

For each static tracepoint, LTTng defines a function that is used to extract and record the trace static data. These functions can replace the get_##name##_kgtp_string probes used in the current implementation, and also the KGTP probe that stores the data, in case we want to collect only the tracepoint static data. The algorithm used at the tracepoint site becomes:

```
create the kgtp_event_call structure
   if (tracepoint_enabled)
      if (need_to_collect_the_registers)
        save registers in the pt_regs structure
      if (no_condition OR condition_is_true)
         call the old trace function
```

In that case, KGTP is in charge of generating the tracepoint condition native code and evaluating it using the registers collected. By registering the LTTng function to the tracepoint, it will be called by the old trace function. Based on the results presented in Table **4**, and assuming that evaluating the true condition (2*counter1+3*counter2>0) takes the same time as a false condition (2*counter1+3*

counter2<0), we can conclude that the time needed to evaluate the registers and to evaluate the condition at the tracepoint site is equal to the execution time presented in Table **4** in the case we have a false condition, which is 154 cycles. We can then extrapolate these results to measure the new execution time. The equation above becomes:

T = Treg + Tcond + Tlttng

Treg + Tcond is the time needed to collect the registers and to evaluate the condition by KGTP and Tlttng is the time taken by LTTng to collect the tracepoint data. Tlttng was measured on a vanilla kernel where we defined the same static tracepoint used for the other test cases. The same test module was used to make the measurements. Table **7** presents the extrapolated results.

Table 7. KGTP-LTTng Integration

	KGTP+LTTng (Cycles)	Systemtap (Cycles)
Condition	154	223
Data	333	1252
Condition and data	487	1336

With this implementation, the call to the original trace function is performed only if the KGTP condition is true. Therefore, this mechanism can be used by all the tracing tools that can register to TRACE_EVENT and is not limited to LTTng.

Integrating KGTP with LTTng lets us benefit from the conditional dynamic and static tracing capabilities of KGTP and the great performance of LTTng ring buffers. Therefore, this solution is more flexible than Systemtap and, at the same time, offers better performance.

5. CONCLUSION

We have described an implementation based on the existing KGTP kernel module and GDB that offers conditional dynamic and static tracepoints. Conditions are defined using complex C-like expressions that can use all the variables accessible from the tracepoint address. All the arithmetic and logic operations are supported by KGTP. Both dynamic and static tracepoints are able to evaluate and save the values of user-defined expressions similar to those used in the conditions.

Additionally, the tool is able to collect static tracepoints data as defined by the TRACE_EVENT macro. Unlike Systemtap, our implementation is able to extract the data manually without the need to redefine that data. With the ability of inserting dynamic tracepoints or reusing static tracepoints, and to specify arbitrary conditions and data collection expressions, our initial objectives have been achieved.

Even though we showed that our implementation is faster than Systemtap for dynamic tracepoints and has comparable execution times for static tracepoints, we suffered from the low performance of KGTP buffering scheme and some optimizations are required in order to reach the performance of other tools such as Ftrace and LTTng.

The bytecode produced by GDB, for the expressions used in the conditions and actions, could easily be further optimized by eliminating unneeded operations.

Static data extraction can also be optimized. Instead of generating and copying strings into the trace buffers, we can simply save the intermediate structure used to extract the data and use it to generate the string only when the user is viewing the trace.

Finally, the results also show that the data structures used to store the trace are not optimized and can be replaced by more efficient structures like ring buffers.

ACKNOWLEDGEMENT

The financial support of NSERC, Defence Research and Development Canada and Ericsson Research is gratefully acknowledged.

CONFLICT OF INTEREST

Declared none.

REFERENCES

[1] “Tracingbook.” 2012. [Online] Available: http://lttng.org/tracingwiki/index.php/TracingBook
[2] D. Toupin, "Using tracing to diagnose or Monitor Systems", *Softw. IEEE.,* vol. 28, pp. 87-91, 2011.
[3] S. Rostedt, “Using the TRACE_EVENT() macro,” 2010. Available from: http://lwn.net/Articles/379903/
[4] UST Official Website. 2012. Available: http://lttng.org/ust
[5] R. Pesch, R. Stallman, and S. Shebs, *Debugging with GDB: the GNU Source-Level Debugger for GDB*. Free Software Foundation:USA 2010.
[6] R. Pesch, R. Stallman,and S. Shebs, “ Bytecode Descriptions,” Available: http://sourceware.org/gdb/current/onlinedocs/gdb/Bytecode-Descriptions.html#Bytecode-Descriptions [Accessed: 16th Jan. 2012]
[7] R. Pesch, R. Stallman, and S. Shebs, “General Bytecode Design,” Available: http://sourceware.org/gdb/current/onlinedocs/gdb/General-Bytecode-Design.html#General-Bytecode-Design [Accessed: 16th Jan.2012].
[8] “KGTP Patch,” 2011. Available: http://lwn.net/Articles/430666/
[9] S. Goswami, “An introduction to Kprobes,” *Linux Wkly News*, 2005. Available: http://lwn.net/Articles/132196/[Accessed: 16th Jan.2012].
[10] “Kprobes Documentation in the Kernel,” Available: http://lxr.free-electrons.com/source/Documentation/kprobes.txt [Accessed: 16th Jan. 2012].
[11] P. Panchamukhi, A. Mavinakayanahalli, J. Keniston, A. Keshavamurthy, and M. Hiramatsu, “Probing the Guts of Kprobes,” In: *Ottawa Linux Symposium*, Ottawa, Canada, 2006.
[12] “Dtrace page in the Tracing Wiki,” “Available: http://lttng.org/tracingwiki/index.php/Dtrace [Accessed: 16th Jan 2012].
[13] B. M. Cantrill, M. W. Shapiro and A. H. Leventhal, "Dynamic Instrumentation of production systems," In: *USENIX Annual Technical Conference*, Boston, MA: USA, 2004.
[14] J. Corbet, “On DTrace Envy ,” 2007. Available: http://lwn.net/Articles/244536/ [Accessed: 16th Jan.2012].
[15] “Debug Exceptions, ” Available: http://www.logix.cz/michal/doc/i386/chp12-03.htm/[Accessed:16th Jan.2012]
[16] J. Corbet, “Tracing: no shortage of options,” 2008. Available: http://lwn.net/Articles/291091//[Accessed:16th Jan.2012]
[17] F. Ch. Eigler, V. Prasad, W. Cohen, H. Nguyen, M. Hunt, J. Keniston and B. Chen, "Architecture of systemtap: a Linux trace/probe tool," 2005. [Online] Available from: http://sourceware.org/ systemtap/archpaper.pdf
[18] F. C. Eigler, "Problem solving with systemtap," In: *Red Hat Summit* 2007, San Diego, 2007.

[19] "Systemtap Official Website," Available: http://sourceware.org/systemtap/[Accessed: 16th Jan.2012]

[20] "Ftrace documentation in the Kernel," Available: http://lxr.free-electrons.com/source/Documentation/trace/ftrace.txt[Accessed:16th Jan.2012]

[21] S. Rostedt, "Ftrace: Now and Then," 2010. Available: http://lttng.org/tracingwiki/images/e/e6/RostedtLinuxCon2010.pdf[Accessed:16th Jan.2012]

[22] J. Corbet, "Dynamic probes with ftrace," 2009. Available: http://lwn.net/Articles/343766/[Accessed:16th Jan.2012]

[23] "Ftrace Dynamic Tracepoints Documentation," Available: http://lxr.free-electrons.com/source/Documentation/trace/kprobetrace.txt [Accessed:16th Jan.2012].

[24] R. W. Wisniewski, R. Azimi, M. Desnoyers, M. M. Michael, J. Moreira, D. Shiloach, and L. Soares, "Experiences understanding performance in a commercial scale-out environment," In: *International Euro-Par Conference*, Rennes: France, 2007.

[25] M. Bligh, M. Desnoyers, and R. Schultz, "Linux kernel debugging on google-sized clusters," In: *Ottawa Linux Symposium*, Ottawa, Ontario: Canada, 2007.

Received: February 24, 2012 Revised: April 14, 2012 Accepted: April 16, 2012

© Fahem and Dagenais; Licensee *Bentham Open*.

This is an open access article licensed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted, non-commercial use, distribution and reproduction in any medium, provided the work is properly cited.