PACT: U: An Event-Based Language for Dynamic Binary Translation Frameworks

Serguei Makarov
University of Toronto
serhei@cs.toronto.edu

Angela Demke Brown
University of Toronto
demke@cs.toronto.edu

Ashvin Goel
University of Toronto
ashvin@eecg.toronto.edu

Abstract. Dynamic binary translation frameworks provide a powerful way of instrumenting programs for analysis. However, writing new analysis tools based on a DBT framework is excessively difficult due to the need to describe a transformation over the target program at an assembly-code level. We present EBT, a domain-specific language for writing DBT client modules inspired by the event-based model of the SystemTap and DTrace languages. A script in the EBT language describes a set of events within the target program, along with handler code to execute in response to each event. Events are specified using a general model consisting of a small set of primitives combined with a mechanism to compute additional predicates over data available from the context of the event. The EBT language translator resolves these high-level events to an exact description of how to instrument each location in the target program, producing C source code for an analysis tool based on the DynamoRIO DBT framework.

1 Introduction

Developers studying the behaviour of their programs and trying to diagnose problems may employ a wide variety of analysis tools to gather information. Analysis can be performed on an already-compiled target program by placing breakpoints and instrumentation through interfaces such as `uprobes` [1], `ptrace` [2] or DynInst [3], or by running the program in a controlled environment such as the ones provided by Valgrind or DynamoRIO [4]. The latter frameworks can handle some of the book-keeping necessary to rewrite the entirety of the target program, allowing applications such as watchpoints or instruction-level profiling. Writing a new analysis tool based on such a framework requires a significant time investment, as the analysis must be described in terms of a low-level transformation over the target program's machine code.

The developer may need to specify how to traverse machine code to identify sections that match a certain predicate or to perform appropriate safety checks before reading data from memory. Busy engineers may not be willing to invest time in learning how to translate their questions into the model provided by an unfamiliar framework, even if it might technically support the exact type of analysis they require. Therefore, the usefulness of an instrumentation framework is often evaluated by the usefulness of the pre-written tools that ship with it, while the full power and flexibility of the underlying model is something that will only be taken advantage of by a handful of more determined developers. However, a lot of the work involved in writing analysis tools is predictable enough to be automated.

In recent years, frameworks such as SystemTap [5] and DTrace [6] have emerged, providing *event-based instrumentation* (EBI) languages that expose the observation facilities of a lower-level framework through a uniform and easy-to-learn interface. An event-based instrumentation model consists of the following conceptual elements, all of which are either already familiar to a programmer or easy to understand:

- A set of high-level events that can be observed in the target system. For example, the SystemTap standard libraries include a "context switch" event `kernel.ctxswitch` which triggers on every context switch performed by the operating system. The in-kernel interfaces used to instrument the event are abstracted away from the user.
- Data from the context of the event in the target program. For example, context data provided by the SystemTap `kernel.ctxswitch` event includes the PIDs and names of the processes that the kernel is currently switching between.
- A generic scripting language for describing event handlers. Thus, whenever an event occurs, the script writer can specify an action to take using the available context data. For example, logging a message, or tracking a statistic (e.g. a timestamp or a running average) using standard data structures such as counters and associative arrays.

As long as a user can describe their analysis in terms of a set of events to observe and react to, they will be able to quickly figure out how to write a corresponding script in an event-based instrumentation language. A large variety of profiling- and tracing-type analysis tools can be expressed using this type of model.

However, the existing EBI frameworks can only be used for non-disruptive observations. The DTrace language and event model is limited to describing fixed-length tracing instrumentations that can be handled by a highly restricted in-kernel API [7]. The SystemTap framework provides a more flexible language and run-time model, but it is still limited to describing passive observations of a system. These limitations amount to a deliberate safety guarantee, which makes it feasible to use SystemTap- and DTrace-based tools to diagnose problems on production machines, without disrupting the system's operation or performance [8]. Moreover, to maintain their safety and performance guarantees, SystemTap and DTrace are limited to lightweight instrumentation methods such as static tracepoints, breakpoint APIs, and trampoline-based instrumentation frameworks along the lines of DynInst.

None of these guarantees are relevant to instrumentation-based analyses performed in a development environment, where the programmer may instead prefer to have more information available as well as more control over the program's execution at the possible cost of performance.

Event-based instrumentation has potential applications to more heavyweight or intrusive types of instrumentation, such as those provided by dynamic binary translation (DBT) frameworks. By rewriting the program dynamically as each new unit of code is scheduled for execution, DBT frameworks such as Pin [9] or DynamoRIO [4] support pervasive analyses that gather additional types of information all across the target program. To expose these instrumentation models through an event-based language, new language elements are necessary – in particular, in the design of the model used for specifying events and extracting context data from them.

This paper explores the development of high-level, event-based models for writing program analysis tools. Based on our model we present EBT, a prototype language for describing analysis tools that use the DynamoRIO DBT framework.

2 Language Design

An EBT user writes their analysis tool in the form of a *script program*, which can include declarations specifying *probes*, *global data*, and *functions*.

Probes describe instrumentation that is to be inserted into the target program. Global data is used to store information collected by the probes, either in the script's local memory or as *shadow memory values* associated to specific address locations in the target program's memory, managed by a framework such as Umbra [10]. Functions can be called from elsewhere in the script as in any high-level scripting language. Figure 1 shows a sample EBT script which demonstrates all of these elements. It can be run on a target program such as the one in Figure 2 to profile division instructions whose divisor operand is a power of 2.

```
array divs    // - divisions
array divs_p2 // — divisions by a power of 2

func is_power_of_2(n, name) {
  return (n & (n-1)) == 0
}

/* Count the integer divisions performed
   in each function of the target program: */
probe insn($opcode == "div" || $opcode == "idiv")
      and function {
  divs[$name]++
  // On x86, the divisor is the first operand:
  if (is_power_of_2(@op[0])) divs_p2[$name]++
}

/* Once the program finishes running,
   print a summary: */
probe end {
  printf("  div | div_p2 | function\n")
  for (fname in div_count)
    printf("%6d | %6d | %s\n", divs[fname],
      divs_p2[fname], fname)
}
```

Figure 1: `insn-profiler.ebt`, a simple script in the EBT language that counts integer divisions by a power of 2.

The instrumentation performed by an EBT script is described as a series of probes. Each probe consists of an *event expression* along with an associated *handler code* block. This follows a clear separation of concerns: the handler code is used to describe *what* additional code to insert into the target program, while the event expression identifies *when* that code should run. The answer to this question of 'when?' may be as simple as 'a specific point in the target program's code', or it may require computing a complex series of predicates to satisfy a query such as 'every instruction matching a certain predicate' or 'every access to a certain memory object'. As a secondary concern, the event expression also describes *what data from the target program is available* when the handler code runs. Data which can be obtained from within the target program at the point where the event occurs is called *context data*.

This separation is also a good reflection of the internal logic of a DBT client. A client for a DBT framework must include code to identify where to instrument the target program as well as code that will actually be injected into the target program. Computations relating to the former task can be described as running during '*instrumentation-time*', while computations occurring in injected code can be described as happening during the target program's '*run-time*'. Event expressions in EBT correspond to instrumentation-time code, while handler blocks execute at runtime. The language used inside the

Target program `test.c`:

```
#include <stdio.h>

int calculate(int y) {
  int i, j; int total = 0;
  for (i = 1; i <= y; i++)
    for (j = 1; j <= i; j++)
      total += i/j;
  return total;
}

int main() {
  printf("result is %d\n", calculate(2350/2));
}
```

Result of running `ebt`:

```
$ ebt insn-profiler.ebt -- test
result is 4650055
    div | div_p2 | function
 690900 |  10889 | calculate
      4 |      0 | do_lookup_x
      2 |      2 | _IO_new_file_xsputn
```

Figure 2: Result of running `insn-profiler.ebt` on a simple C target program on x86. Observe that the division in `main()` has been optimized away and that the standard C runtime performs additional divisions.

handler blocks is a simple scripting language inspired by the SystemTap language [11]. It supports control-flow, function calls, and allows access to global data structures such as associative arrays.

Our key design challenge was to provide an event description language general enough to specify the kinds of instrumentation commonly performed by DBT clients, while retaining fairly obvious performance guarantees and allowing for readable script programs. In particular, there should be a clear distinction between code which runs at instrumentation-time and code which actually executes during the program's runtime – the latter results in a far greater performance penalty to the target program [4]. It should also be easy to determine which data from the target program is available to use inside each of the handler blocks.

The event description language in EBT pertains entirely to instrumentation-time computation. Event expressions are composed using *combinators* out of *basic events* and *context specifiers*, each of which may include *predicates*.

Basic Events. Each *basic event* provided by EBT refers to a category of events occurring in the target program. Examples of basic events include `insn` ('every instruction') and `function.entry` ('every entry point to a function'). A table of basic events, along with their associated meanings, is given in Figure 3. Each basic event defines a corresponding set of *context values* which provide further information about the event occurrence to the subsequent handler block as well as to any predicates in the event expression (see below). Thus, for an `insn` event, it is possible to access the opcode `$opcode` as well as input (source) operands `@op[1]`, `@op[2]`, ... and output (destination) operands `@out[1]`, ... of the instruction.

Basic events:

Basic Event	Meaning	Context Data
`insn`	instruction	`$opcode`, *operands...*
`function.entry`	function call	`$name`, *arguments...*
`function.exit`	function return	`$name`, `@retval`
`obj.alloc`	`malloc()`	`@addr`, `@size`
`obj.access`	memory access	`@addr`

Specifiers:

Specifier	Meaning	Context Data
`function`	current function	`$name`, *local variables...*
`thread`	current thread	`@id`

Figure 3: Basic events and specifiers provided by EBT.

Context values are distinguished according to whether they can be inferred from the target program's code at instrumentation-time or whether they can only be obtained at run-time. EBT makes the distinction explicit by prefixing the names of instrumentation-time context values with a '`$`' sigil, and runtime values with an '`@`' sigil.

Context Specifiers. When the context values provided by a basic event do not include necessary data, additional context can be obtained using *specifiers*. For example, the `function` specifier imports context information about the current function. By itself, a specifier does not suggest a location to instrument; in order to form a valid event expression it must be combined with a basic event. For example, '`insn and function`' is a valid event expression that refers to every instruction execution in the target program and also imports information about the current function. Through the use of specifiers, the script makes explicit which context data is being made available to each handler. This knowledge may be relevant to a script writer when the underlying implementation requires obtaining the data via a complex computation that may cause additional overhead.

Predicates. Predicate expressions are placed in brackets '`()`' after an event expression, and cause the DBT framework to only place instrumentation at locations for which the expression inside the predicate evaluates to '`true`'. Predicate expressions are capable of performing arbitrarily complex computation on these values by invoking functions elsewhere in the script. Judicious use of predicates is important for ensuring that the DBT framework only inserts instrumentation relevant to the desired analysis. For instance, we may only be interested in profiling instructions inside a certain function; in that case, we can write our event as follows:

```
insn ($opcode == "div")
and function ($name == "calculate")
```

Only instrumentation-time context values can be used inside a predicate. Conditions that require runtime values must be checked inside the handler block. For instance, data watchpoints must currently be formulated using a shadow memory field to track whether a memory location is being watched, then checking the address inside the handler block for an `obj.access` event:

```
// Use 1 shadow bit per memory word:
shadow watched:1
array watch_addr

probe obj.alloc and
function ($name == "create_widget") {
  watched[@addr] = true
  // Set an entire range to refer to @addr:
  watch_addr[@addr..@addr+@size-1] = @addr
}

probe obj.access and function {
  if (watched[@addr])
    printf("access widget at %p from %s\n",
           watch_addr[@addr], $name)
}
```

Combinators. Besides the '`and`' operation shown above, event expressions can also be combined using '`or`'. Whereas '`and`' can be used to combine a basic event with an arbitrary number of specifiers, '`or`' is used to combine two alternate events: the corresponding handler block is run whenever either one of the alternate events occurs.

We considered implementing various alternate syntaxes on top of this basic combinator model, such as `function ($name == "foo") :: insn`. Here, the `::` operator would take a list of specifiers as its first operand and a basic event as its second operand, and thus the above expression can be read as 'instructions occurring in function `foo()`'. In practice, this syntax amounts to a cosmetic wrapper around `and`, but may be worth implementing to improve usability and make the structure of event expressions more immediately clear.

3 Translator

EBT is implemented as a translator program that converts a script program to source code for a DBT client. The translator can also automatically compile and run the resulting client against a specified target program. The current prototype version of EBT targets the DynamoRIO framework, a highly efficient DBT framework with support for Linux and Windows platforms.

Scripts in earlier instrumentation languages such as SystemTap and DTrace could be used to express fairly simple instrumentation. An event generally described a specific point in the target program, which could be located with the aid of debug information, and patched to run the handler using a breakpoint- or trampoline-based framework, such as Linux `uprobes` [1], or DynInst [3]. In this model there is a close correspondence between the underlying API of the instrumentation framework, and the high-level event model exposed to the user of the language. However, when the backend is a DBT framework, there is a much greater gap between the instrumentation model and the event model. A conceptually simple event such as "every instruction of a particular type" or "every write to a given memory object" might correspond to a complex instrumentation applied at multiple points of the target program.

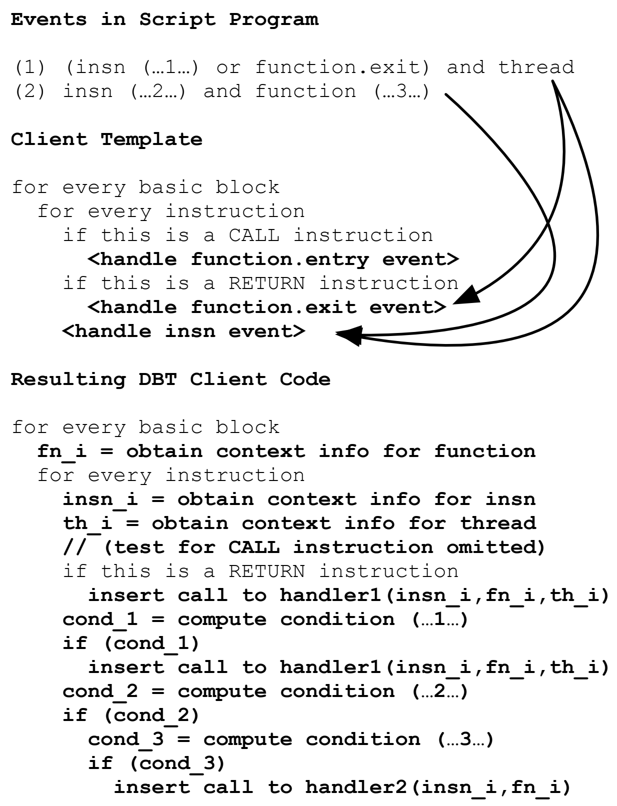
A DBT framework's most basic facility for modifying a target program is a callback which is called for each unit of the target program (typically every basic block [12]) and has the opportunity to examine and rewrite the instructions in the unit. This callback generally runs only once on each unit of code (immediately before the first time the code executes) and is therefore unsuitable for directly capturing events that occur during the program's runtime. Instead, EBT must generate code to examine the basic block and determine which subset of the events specified in the script program occurs at each instruction.

Thus the translation process requires an additional *event resolution* phase. The basic procedure of event resolution is illustrated in Figure 4. Given a set of event expressions, we generate a basic block callback that iterates the instructions in each basic block and checks appropriate predicates for each event to determine whether it occurs at the given instruction.

4 Results and Further Work

The current version of the translator provides a proof-of-concept showing that a language structured around the ideas of events and context data is significantly more effective to work with than the APIs directly provided by a dynamic binary translation framework. As a rule of thumb, what takes hundreds of lines of boilerplate to implement in DynamoRIO takes only ten lines or so of scripting using EBT to accomplish.

This reduction in complexity has a particularly significant impact when the user wants to study their program in an exploratory fashion, changing their analysis repeatedly until they find the information they were looking for. For example, consider the user who needs an instruction profiling tool such as the one implemented in Figure 1. DynamoRIO comes bundled with an example program, `div.c`, which performs a simple aggregate count of `div` instructions across a program, implemented in about 100 lines of code. If the user wants to modify `div.c` to count instructions separately for each function, they must make use of the DrSyms API bundled with DynamoRIO, which requires a rather large amount of boilerplate code to perform a lookup operation and determine which func-



```
Events in Script Program

(1) (insn (…1…) or function.exit) and thread
(2) insn (…2…) and function (…3…)

Client Template

for every basic block
  for every instruction
    if this is a CALL instruction
      <handle function.entry event>
    if this is a RETURN instruction
      <handle function.exit event>
    <handle insn event>

Resulting DBT Client Code

for every basic block
  fn_i = obtain context info for function
  for every instruction
    insn_i = obtain context info for insn
    th_i = obtain context info for thread
    // (test for CALL instruction omitted)
    if this is a RETURN instruction
      insert call to handler1(insn_i,fn_i,th_i)
    cond_1 = compute condition (…1…)
    if (cond_1)
      insert call to handler1(insn_i,fn_i,th_i)
    cond_2 = compute condition (…2…)
    if (cond_2)
      cond_3 = compute condition (…3…)
      if (cond_3)
        insert call to handler2(insn_i,fn_i)
```

Figure 4: Schematic of the event-resolution process, illustrated using an example script and the resulting DBT basic block callback in pseudocode. Code which is generated from the script program events is shown in **boldface**; non-boldface code is part of a fixed template common to all clients.

tion a given instruction belongs to. But symbol lookup is a common enough task which should be much easier to perform in an analysis language. Thus, the comparable functionality in EBT is accessed simply by placing a `function` specifier and using the `$name` context value exported from it. Next, the script writer may want to try profiling a different set of opcodes, or profiling by thread as well as function; or they might want to narrow the analysis down to a small subset of the program – all of these tasks can be accomplished by minor alterations to an EBT script, as opposed to changing dozens of lines of code in a hand-written DynamoRIO client.

Moreover, as various functionality is added to EBT, it can be exposed in a uniform fashion through the context data mechanism, with any underlying setup taken care of on behalf of the user. On the other hand, the various APIs available in DynamoRIO often have differing setup requirements and access methods that must be learned and explicitly dealt with by the user in their client programs.

That said, a lot of additional work is needed for EBT to become a mature and practically useful language. At a minimum, the set of events and specifiers we described in Figure 3 as well as their associated data must be significantly extended to allow access to a fully comprehensive picture of the program's behaviour and make the language's capabilities truly comparable to those of the underlying DynamoRIO framework. The set of datatypes in the EBT language must likewise be extended to allow us to reason about floating-point values and data structures inside the target program. In addition, we would like to formulate and develop language features that would alter the target program's state in addition to observing it, which would allow EBT to also be used for such things as rapid prototyping of automated test cases, development of one-off fault injection tools, and similar debugging applications.

References

[1] Jim Keniston, Ananth Mavinakayanahalli, et al., "Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps," in *Proc. of Ottawa Linux Symposium*, 2007.

[2] Linux manual pages: `ptrace`(2). `http://www.manpages.info/linux/ptrace.2.html`

[3] Brian Buck and Jeffrey K. Hollingsworth, "An API for Runtime Code Patching," in *International Journal of High Performance Computing Applications*, 14 (4) pp. 317-329, Nov 2000.

[4] Derek Bruening, Qin Zhao and Saman Amarasinghe, "Transparent Dynamic Instrumentation," in *Proc. of VEE*, 2012.

[5] Vara Prasad, William Cohen, Frank Ch. Eigler, et al., "Locating System Problems Using Dynamic Instrumentation," in *Proc. of Ottawa Linux Symposium*, 2005.

[6] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal, "Dynamic Instrumentation of Production Systems," in *Proc. of USENIX ATC*, 2004.

[7] See the DTrace manual, chapters 6 and 11, for a description of tracing buffers. `http://dtrace.org/guide/`

[8] The SystemTap Language Reference (as of April 2015). SystemTap Overview: Safety and Security. `http://sourceware.org/systemtap/langref/SystemTap_overview.html`

[9] Chi-Keung Luk, Robert Cohn, et al., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[10] Qin Zhao, Derek Bruening and Saman Amarasinghe, "Umbra: efficient and scalable memory shadowing," in *Proc. of CGO*, 2010.

[11] The SystemTap Language Reference (as of April 2015). `http://sourceware.org/systemtap/langref/Contents.html`

[12] Intel's Pin framework also allows instrumentation to be specified at the level of individual instructions, routines, or entire binary images (i.e. at module load time). See the Pin user guide (as of April 2015): `http://software.intel.com/sites/landingpage/pintool/docs/58423/Pin/html/index.html#GRAN`

[13] DynamoRIO: Symbol Access Library (as of April 2015): `http://dynamorio.org/docs/page_drsyms.html`