An Event-Based Instrumentation Language for Dynamic Binary Translation Systems

Serguei Makarov

University of Toronto
Department of Computer Science – Systems Group

serhei@cs.toronto.edu

June 3, 2014
Analyzing Complex Software is Challenging

How do we study the behaviour of a large, complex software system?

- Debuggers (GDB, LLDB, ...)
- Profiling Tools (OProfile, Gprof, Linux Perf, ...)
- Tracing Tools (LTTng, strace, ...)
- Instrumentation Tools (DynInst, Valgrind, ...)

What if these tools can’t give us the answers we want?

- Need to write a custom program, to get the info we’re interested in.
- Anyone interested in writing a custom kernel module?

There’s easier and more flexible ways....
Event-Based Instrumentation (EBI)

EBI languages (not an official term) are used to quickly develop programs that analyze a running software system by probing a series of events.

Example (DTrace)

```c
# Trace uses of the open() system call.
syscall::open*:entry {
    printf("%s %s",execname,copyinstr(arg0));
}
```

Example (Systemtap)

```c
# Count number of system calls across a Linux system.
global ops

probe syscall.*.return { ops[probefunc()] <<< 1; }
```
# Probe functions listed on command line

```bash
global called

probe kernel.function(@1).call {
  called[probefunc()] <<< 1
}

probe end {
  foreach (fn in called-) # Sort by call count
    printf("%s %d\n", fn, @count(called[fn]))
  exit()
}
```
How does Systemtap work?

- Translate the user’s script into a low-level module.
- Use debug information to locate where events occur inside the program.
- Low-level module inserts *instrumentation code* into the target program using a breakpoint- or trampoline- type instrumentation framework (such as kprobes, uprobes, or DynInst).
- The instrumentation code runs event handlers from the script.
Let’s consider what SystemTap is good for:

- **Major advantages:**
  - Simple, high-level and non-intrusive
  - Minimal effect on performance
  - Can combine information from multiple programs

- **Major limitation:** cannot probe pervasive events:
  - ”Every time we execute an opcode…” → 1000’s of locations
  - ”Every time we access a memory location…” → 10,000’s of locations
DBT framework takes over the execution of a target program. Pass all code through a *client module* before it runs. The client can arbitrarily rewrite the code to introduce instrumentation in as many places as it wants. We can capture pervasively occurring events, such as:

- **Instruction counts**: tally instructions of a particular type
- **Memory watchpoints**: capture memory accesses

Existing frameworks:
- DynamoRIO
- Pin (by Intel)
- Granary (in kernel-mode)
- Valgrind
Example (Count DIV instructions, about 100 lines of code)

// obviously 100 loc doesn’t fit on this slide

static int div_count = 0, div_p2_count = 0;
static void *count_mutex; // multithread support

DR_EXPORT void dr_init(client_id_t cid) { ... }
static dr_emit_flags_t bb_event (  
    void* drcontext, void *tag, instrlist_t *bb, bool for_trace, bool translating  
) { ... } // called for every basic block
static void callback(void) { ... } // increments counters
static void exit_event(void) { ... } // prints summary

This is div.c, one of the examples included with DynamoRIO.
Example (Count DIV instructions, less than 10 lines of code)

```javascript
var div_count = 0, div_p2_count = 0

probe insn ($opcode == "div") {
    div_count ++
    if ((@op2 & (@op2 - 1)) != 0) div_p2_count++
}

probe end {
    printf ("executed %d div instructions, ",
            "%d divided by powers of 2\n",
            div_count, div_p2_count)
}
```
Architecture and Design

- Distinguish instrumentation-time and runtime in the language design.
  - Instrumentation-time: when our DBT client module rewrites the code
  - Runtime: when the target program runs the code
- Resolve events to basic mechanisms:
  - Every instruction
  - Function entries, exits
  - Memory watchpoints
- Generate a client module with **only** the mechanisms we need, to:
  - Check instrumentation-time conditions
  - Insert code to run the event handlers
It Gets Complicated

Serguei Makarov (UofT)  An Event-Based Language for DBTs  June 3, 2014  11 / 14

### Derived Probes

- (FCALL + things we need to check & handle out)
- (INSN +)
- (INSN *)

### Script Probes

- function (...) and insn (...) \& ... \3
- function call (...) or insn (...) \& ... \3

### Client

```plaintext
bb_callback() {
  // If it's a call insn
  do instrumentation for FCALL-type probes
  // Do instrumentation for INSN-type probes

  if (this is a function entry block)
  do inst for FENTRY-type probe
```

- Either
  - manually emit code pre-probe
  - iterate through some kind of table
EVENT RESOLUTION

struct condition {
  can be shared
  (to avoid building multiple copies of the same thing)
  struct handlers
  struct basic-prbe
    - mechanism
      (FCALL, RETURN, INSN, ...)
      - vector condition
      - condition
      - handler

  map (basic-prbe-type, vector (basic-prbe *) )

FCALL
INSN

void regular_prbe ...

map (basic-prbe-type, vector (basic-prbe *) )

enum basic-prbe-type ...

TYPES OF EVENTS

can provide:
  - context info
  - implicit conditions
  - maybe basic probe mechanism

Condition

RESOLUTION PROCESS

1. brute force search through all possible combinations of w's
2. collect conditions (implicit or explicit)
   LEFT TO RIGHT; INNER TO OUTER
3. ensure exactly 1 mechanism

Pretty sure the sharing
will not get too complicated,
but good to know we can support it...
Event Based Instrumentation languages are an underexplored field.

- **What is the problem we want to solve?** We want to ask questions to study the behaviour of a complex software system.

- **Why are EBIs such as Systemtap interesting?** They give programmers a familiar way to formulate unconventional questions.

- **What are Dynamic Binary Translation frameworks?** A powerful way to add instrumentation to a program, by rewriting the code in it.

- **Why do we want to build an EBI around a DBT framework?**
  - We can instrument fine-grained events not available to a tool like Systemtap.
  - We have more potential options to modify the behaviour of the program (for debugging-type tasks, for experiments).
In Conclusion – Our Cast of Characters

Existing frameworks mentioned during this talk:

- SystemTap – http://sourceware.org/systemtap
- DTrace – http://dtrace.org/blogs/about
- DynamoRIO – http://dynamorio.org
- Granary – http://www.granarydbt.org/

Special thanks to:

- My current supervisor: Prof. Angela Demke-Brown.
- My current graduate mentor: Peter Goodman.
- Folks at Red Hat for hiring me to work with SystemTap in 2012-2013.
- Other people who helped me get into research, in all kinds of ways: Prof. Karen Reid, Prof. Faith Ellen.