Sample Use Cases

We provide several examples in the `samples` directory of the release package to illustrate how the DynamoRIO API is used to build a DynamoRIO client.

There are also samples for the Dr. Memory Framework located in the `drmemory/drmf/samples` directory of the release package.

For larger examples of clients, see the provided **DynamoRIO-Based Tools**, which are larger and more polished end-user clients than these samples.

List of Samples

The sample bbbuf.c demonstrates how to use a TLS field for per-thread basic block profiling.

The sample bbcount.c illustrates how to perform performant instrumentation for reporting the dynamic execution count of all basic blocks.

The sample bbsize.c collects statistics on the sizes of all basic blocks in the target application.

The sample cbr.c collects conditional branch execution information and shows how to dynamically update or replace instrumented code after it executes.

The sample cbrtrace.c collects conditional branch execution traces and writes them into files.

The sample countcalls.c reports the dynamic execution count for direct calls, indirect calls, and returns in the target application. It illustrates how to perform performant inline increments and use per-thread data structures.

The sample div.c demonstrates profiling the types of values fed to a particular opcode.

The sample empty.c is provided as an example client that does nothing.

The sample inc2add.c performs a dynamic optimization: it converts the "inc" instruction to "add 1" without perturbing the target application's behavior.

The sample inline.c performs an optimization that uses the custom trace API to inline entire callees into traces.

The sample inscount.c reports the dynamic count of the total number of instructions executed via inserting performant clean calls which are auto-inlined by DynamoRIO.

The sample instrcall.c demonstrates how to instrument direct calls, indirect calls and returns.

The sample memtrace.c is provided as an example client that illustrates how to create a private code

cache and perform lean procedure calls.

The sample modxfer.c reports the control flow transfers between modules.

The sample modxfer_app2lib.c reports the control flow transfers between the application executable and other dynamic libraries and modules. It illustrates how to perform performant clean calls on different modules.

The sample opcodes.c computes dynamic execution counts broken down by instruction opcode.

The sample prefetch.c demonstrates modifying the dynamic code stream for compatibility between different processor types.

The sample signal.c demonstrates how to use the signal event.

The sample stl_test.c is provided as an example client that uses C++ STL containers.

The sample syscall.c displays how to use the system call events and API routines.

The sample tracedump.c is provided as a standalone application that disassembles a trace dump in binary format produced by the -tracedump_binary option.

The sample wrap.c demonstrates how to use the drwrap extension (see **Function Wrapping and Replacing Extension**).

The sample ssljack.c demonstrates how to hook OpenSSL and GnuTLS functions, using the drwrap extension.

Discussion of Selected Samples

Instruction Counting

We now illustrate how to use the above API to implement a simple instrumentation client for counting the number of executed call and return instructions in the input program. Full code for this example is in the file samples/countcalls.c.

The client maintains set of three counters: num_direct_calls, num_indirect_calls, and num_returns to count three different types of instructions during execution. It keeps both thread-private and global versions of these counters. The client initializes everything by supplying the following `dr_client_main` routine:

```
DR_EXPORT void
dr_client_main(client_id_t id, int argc, const char *argv[])
{
  /* register events */
  dr_register_exit_event(event_exit);
  dr_register_thread_init_event(event_thread_init);
  dr_register_thread_exit_event(event_thread_exit);
  dr_register_bb_event(event_basic_block);
```

```
  /* make it easy to tell, by looking at log file, which client executed */
  dr_log(NULL, LOG_ALL, 1, "Client 'countcalls' initializing\n");
}
```

The client provides an event_exit routine that displays the final values of the global counters as well as a thread_exit routine that shows the counter totals on a per-thread basis.

The client keeps track of each thread's instruction counts separately. To do this, it creates a data structure that will be separately allocated for each thread:

```
typedef struct {
  int num_direct_calls;
  int num_indirect_calls;
  int num_returns;
} per_thread_t;
```

Now the thread hooks are used to initialize the data structure and to display the thread-private totals :

```
static void event_thread_init(void *drcontext)
{
  /* create an instance of our data structure for this thread */
  per_thread *data = (per_thread *)
    dr_thread_alloc(drcontext, sizeof(per_thread));
  /* store it in the slot provided in the drcontext */
  dr_set_tls_field(drcontext, data);
  data->num_direct_calls = 0;
  data->num_indirect_calls = 0;
  data->num_returns = 0;
  dr_log(drcontext, LOG_ALL, 1, "countcalls: set up for thread "TIDFMT"\n",
         dr_get_thread_id(drcontext));
}

static void event_thread_exit(void *drcontext)
{
  per_thread *data = (per_thread *) dr_get_tls_field(drcontext);

  ... // string formatting and displaying

  /* clean up memory */
  dr_thread_free(drcontext, data, sizeof(per_thread));
}
```

The real work is done in the basic block hook. We simply look for the instructions we're interested in and insert an increment of the appropriate thread-local and global counters, remembering to save the flags, of course. This sample has separate paths for incrementing the thread private counts for shared vs. thread-private caches (see the -thread_private option) to illustrate the differences in targeting for them. Note that the shared path would work fine with private caches.

```
static void
insert_counter_update(void *drcontext, instrlist_t *bb, instr_t *where, int offset)
{
  /* Since the inc instruction clobbers 5 of the arithmetic eflags,
   * we have to save them around the inc. We could be more efficient
   * by not bothering to save the overflow flag and constructing our
   * own sequence of instructions to save the other 5 flags (using
   * lahf) or by doing a liveness analysis on the flags and saving
   * only if live.
   */
  dr_save_arith_flags(drcontext, bb, where, SPILL_SLOT_1);
```

```
  /* Increment the global counter using the lock prefix to make it atomic
   * across threads. It would be cheaper to aggregate the thread counters
   * in the exit events, but this sample is intended to illustrate inserted
   * instrumentation.
   */
  instrlist_meta_preinsert(bb, where, LOCK(INSTR_CREATE_inc
    (drcontext, OPND_CREATE_ABSMEM(((byte *)&global_count) + offset, OPSZ_4))));

  /* Increment the thread private counter. */
  if (dr_using_all_private_caches()) {
    per_thread_t *data = (per_thread_t *) dr_get_tls_field(drcontext);
    /* private caches - we can use an absolute address */
    instrlist_meta_preinsert(bb, where, INSTR_CREATE_inc(drcontext,
        OPND_CREATE_ABSMEM(((byte *)&data) + offset, OPSZ_4)));
  } else {
    /* shared caches - we must indirect via thread local storage */
    /* We spill xbx to use a scratch register (we could do a liveness
     * analysis to try and find a dead register to use). Note that xax
     * is currently holding the saved eflags. */
    dr_save_reg(drcontext, bb, where, REG_XBX, SPILL_SLOT_2);
    dr_insert_read_tls_field(drcontext, bb, where, REG_XBX);
    instrlist_meta_preinsert(bb, where,
        INSTR_CREATE_inc(drcontext, OPND_CREATE_MEM32(REG_XBX, offset)));
    dr_restore_reg(drcontext, bb, where, REG_XBX, SPILL_SLOT_2);
  }

  /* restore flags */
  dr_restore_arith_flags(drcontext, bb, where, SPILL_SLOT_1);
}

static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{
  instr_t *instr, *next_instr;

  ... // some logging

  for (instr = instrlist_first(bb); instr != NULL; instr = next_instr) {
    /* grab next now so we don't go over instructions we insert */
    next_instr = instr_get_next(instr);

    /* instrument calls and returns -- ignore far calls/rets */
    if (instr_is_call_direct(instr)) {
      insert_counter_update(drcontext, bb, instr,
                            offsetof(per_thread_t, num_direct_calls));
    } else if (instr_is_call_indirect(instr)) {
      insert_counter_update(drcontext, bb, instr,
                            offsetof(per_thread_t, num_indirect_calls));
    } else if (instr_is_return(instr)) {
      insert_counter_update(drcontext, bb, instr,
                            offsetof(per_thread_t, num_returns));
    }
  }

  ... // some logging

  return DR_EMIT_DEFAULT;
}
```

Building the Example

For general instructions on building a client, see **Building a Client**.

To build the `instrcalls.c` client using CMake, if `DYNAMORIO_HOME` is set to the base of the DynamoRIO release package:

```
mkdir build
cd build
cmake -DDynamoRIO_DIR=$DYNAMORIO_HOME/cmake $DYNAMORIO_HOME/samples
make instrcalls
```

To build 32-bit samples when using gcc with a default of 64-bit, use:

```
mkdir build
cd build
CFLAGS=-m32 CXXFLAGS=-m32 cmake -DDynamoRIO_DIR=$DYNAMORIO_HOME/cmake
       $DYNAMORIO_HOME/samples
make instrcalls
```

The result is a shared library instrcalls.dll or libinstrcalls.so. To invoke the client library, follow the instructions under **Deployment**.

Instruction Profiling

The next example shows how to use the provided control flow instrumentation routines, which allow more sophisticated profiling than simply counting instructions. Full code for this example is in the file samples/instrcalls.c.

As in the previous example, the client is interested in direct and indirect calls and returns. The client wants to analyze the target address of each dynamic instance of a call or return. For our example, we simply dump the data in text format to a separate file for each thread. Since FILE cannot be exported from a DLL on Windows, we use the DynamoRIO-provided file_t type that hides the distinction between FILE and HANDLE to allow the same code to work on Linux and Windows. We make use of the thread initialization and exit routines to open and close the file. We store the file for a thread in the user slot in the drcontext.

```
static void event_thread_init(void *drcontext)
{
  /* we're going to dump our data to a per-thread file */
  file_t f;
  char logname[512];

  ... // filename generation

  f = dr_open_file(fname, false/*write*/);
  DR_ASSERT(f != INVALID_FILE);

  /* store it in the slot provided in the drcontext */
  dr_set_tls_field(drcontext, (void *)f);

  ... // logging
}

static void event_thread_exit(void *drcontext)
{
  file_t f = (file_t)(ptr_uint_t) dr_get_tls_field(drcontext);
  dr_close_file(f);
}
```

The basic block hook inserts a call to a procedure for each type of instruction, using the API-provided dr_insert_call_instrumentation and dr_insert_mbr_instrumentation routines, which insert calls to procedures with a certain signature.

```
static dr_emit_flags_t
event_basic_block(void *drcontext, void *tag, instrlist_t *bb,
                  bool for_trace, bool translating)
{
  instr_t *instr, *next_instr;

  ... // logging

  for (instr = instrlist_first(bb); instr != NULL; instr = next_instr) {
    next_instr = instr_get_next(instr);
    if (!instr_opcode_valid(instr))
        continue;
    /* instrument calls and returns -- ignore far calls/rets */
    if (instr_is_call_direct(instr)) {
        dr_insert_call_instrumentation(drcontext, bb, instr, (app_pc)at_call);
    } else if (instr_is_call_indirect(instr)) {
        dr_insert_mbr_instrumentation(drcontext, bb, instr, (app_pc)at_call_ind,
                                      SPILL_SLOT_1);
    } else if (instr_is_return(instr)) {
        dr_insert_mbr_instrumentation(drcontext, bb, instr, (app_pc)at_return,
                                      SPILL_SLOT_1);
    }
  }
  return DR_EMIT_DEFAULT;
}
```

These procedures look like this :

```
static void
at_call(app_pc instr_addr, app_pc target_addr)
{
  file_t f = (file_t)(ptr_uint_t) dr_get_tls_field(dr_get_current_drcontext());
  dr_mcontext_t mc;
  dr_get_mcontext(dr_get_current_drcontext(), &mc, NULL);
  dr_fprintf(f, "CALL @ "PFX" to "PFX", TOS is "PFX"\n",
             instr_addr, target_addr, mc.xsp);
}

static void
at_call_ind(app_pc instr_addr, app_pc target_addr)
{
  file_t f = (file_t)(ptr_uint_t) dr_get_tls_field(dr_get_current_drcontext());
  dr_fprintf(f, "CALL INDIRECT @ "PFX" to "PFX"\n", instr_addr, target_addr);
}

static void
at_return(app_pc instr_addr, app_pc target_addr)
{
  file_t f = (file_t)(ptr_uint_t) dr_get_tls_field(dr_get_current_drcontext());
  dr_fprintf(f, "RETURN @ "PFX" to "PFX"\n", instr_addr, target_addr);
}
```

The address of the instruction and the address of its target are both provided. These routines could perform some sort of analysis based on these addresses. In our example we simply print out the data.

Modifying Existing Instrumentation

In this example, we show how to update or replace existing instrumentation after it executes. This ability is useful for clients performing adaptive optimization. In this example, however, we are interested in recording the direction of all conditional branches, but wish to remove the overhead of instrumentation once we've gathered that information. This code could form part of a dynamic CFG builder, where we

want to observe the control-flow edges that execute at runtime, but remove the instrumentation after it executes.

While DynamoRIO supports direct fragment replacement, another method for re-instrumentation is to flush the fragment from the code cache and rebuild it in the basic block event callback. In other words, we take the following approach:

1. In the basic block event callback, insert separate instrumentation for the taken and fall-through edges.
2. When the basic block executes, note the direction taken and flush the fragment from the code cache.
3. When the basic block event triggers again, insert instrumentation only for the unseen edge. After both edges have triggered, remove all instrumentation for the cbr.

We insert separate clean calls for the taken and fall-through cases. In each clean call, we record the observed direction and immediately flush the basic block using **dr_flush_region()**. Since that routine removes the calling block, we redirect execution to the target or fall-through address with **dr_redirect_execution()**. The file samples/cbr.c contains the full code for this sample.

Optimization

For the next example we consider a client application for a simple optimization. The optimizer replaces every increment/decrement operation with a corresponding add/subtract operation if running on a Pentium 4, where the add/subtract is less expensive. For optimizations, we are less concerned with covering all the code that is executed; on the contrary, in order to amortize the optimization overhead, we only want to apply the optimization to hot code. Thus, we apply the optimization at the trace level rather than the basic block level. Full code for this example is in the file samples/inc2add.c.

Custom Tracing

This example demonstrates the custom tracing interface. It changes DynamoRIO's tracing behavior to favor making traces that start at a call and end right after a return. It demonstrates the use of both custom trace API elements :

```
int query_end_trace(void *drcontext, void *trace_tag, void *next_tag);
bool dr_mark_trace_head(void *drcontext, void *tag);
```

Full code for this example is in the file samples/inline.c.

Use of Floating Point Operation in a Client

Because saving the floating point state is very expensive, DynamoRIO seeks to do so on an as needed basis. If a client wishes to use floating point operations it must save and restore the application's floating point state around the usage. For an inserted clean call out of the code cache, this can be conveniently done using **dr_insert_clean_call()** and passing true for the save_fpstate parameter. It can also be done

explicitly using these routines:

```
void proc_save_fpstate(byte *buf);
void proc_restore_fpstate(byte *buf);
```

These routines must be used if floating point operations are performed in non-inserted-call locations, such as event callbacks. Note that there are restrictions on how these methods may be called: see the documentation in the header files for additional information. Note also that the floating point state must be saved around calls to our provided printing routines when they are used to print floats. However, it is not necessary to save and restore the floating point state around floating point operations if they are being used in the initialization or termination routines.

This example client counts the number of basic blocks processed and keeps statistics on their average size using floating point operations. Full code for this example is in the file samples/bbsize.c.

Use of Custom Client Statistics with the Windows GUI

The new Windows GUI will display custom client statistics, if they are placed in shared memory with a certain name. The sample samples/stats.c gives code for the protocol used in the form of a sample client that counts total instructions, floating-point instructions, and system calls.

Note that the stats.c example client and the Windows GUI must both be run within the same session in order for the statistics to be shared properly. They can be modified to use a "Global" prefix instead of "Local" for cross-session sharing, though this requires running with administrative privileges.

Use of Standalone API

The binary tracedump reader also functions as an example of **IA-32/AMD64/ARM Disassembly Library** : samples/tracedump.c.