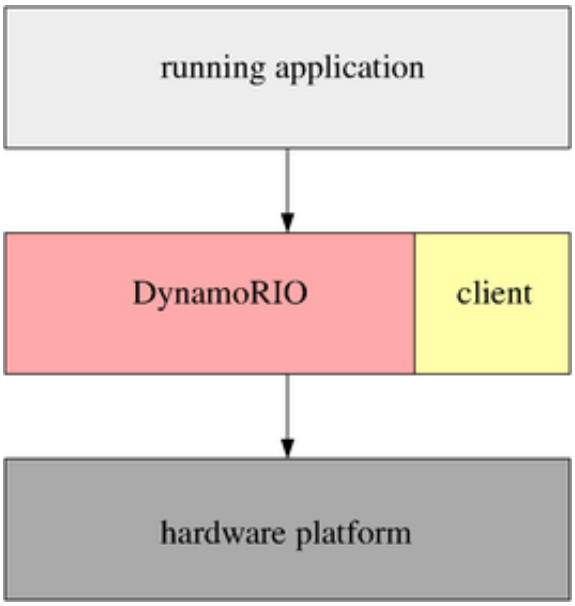
Usage Model for DynamoRIO

This section gives an overview of how to use DynamoRIO, divided into the following sub-sections:

DynamoRIO exports a rich Application Programming Interface (API) to the user for building a DynamoRIO *client*. A DynamoRIO client is a library that is coupled with DynamoRIO in order to jointly operate on an input program binary:



To interact with the client, DynamoRIO provides specific events that a client can intercept. Event interception functions, if supplied by a user client, are called by DynamoRIO at appropriate times.

DynamoRIO can alternatively be used as a third-party disassembly library (see **IA-32/AMD64/ARM Disassembly Library**).

Common Events

A client's primary interaction with the DynamoRIO system is via a set of event callbacks. These events include the following:

- Basic block and trace creation or deletion (**dr_register_bb_event()**, **dr_register_trace_event()**, **dr_register_delete_event()**)
- Process initialization and exit (**dr_client_main()**, **dr_register_exit_event()**)
- Thread initialization and exit (**dr_register_thread_init_event()**, **dr_register_thread_exit_event()**)
- Fork child initialization (Linux-only); meant to be used for re-initialization of data structures and creation of new log files (**dr_register_fork_init_event()**)
- Application library load and unload (**dr_register_module_load_event()**, **dr_register_module_unload_event()**)
- Application fault or exception (signal on Linux) (**dr_register_exception_event()**, **dr_register_signal_event()**)
- System call interception: pre-system call, post-system call, and system call filtering by number (**dr_register_pre_syscall_event()**, **dr_register_post_syscall_event()**, **dr_register_filter_syscall_event()**)
- Signal interception (Linux-only) (**dr_register_signal_event()**)
- Nudge received - see **Communication** (**dr_register_nudge_event()**)

Typically, a client will register for the desired events at initialization in its **dr_client_main()** routine. DynamoRIO then calls the registered functions at the appropriate times. Each event has a specific registration routine (e.g., **dr_register_thread_init_event()**: see the names in parentheses in the list above) and an associated unregistration routine. The header file **dr_events.h** contains the declarations for all registration and unregistration routines.

Note that clients are allowed to register multiple callbacks for the same event. DynamoRIO also supports mutiple clients, each of which can register for the same event. In this case, DynamoRIO sequences event callbacks in reverse order of when they were registered. In other words, the first registered callback receives event notification last. This scheme gives priority to a callback registered earlier, since it can override or modify the actions of clients registered later. Note that DynamoRIO calls each client's **dr_client_main()** routine according to the client's priority (see **Multiple Clients** and **dr_register_client()** in the deployment API).

Systems registering multiple callbacks for a single event should be aware that client modifications are visible in subsequent callbacks. DynamoRIO makes no attempt to mitigate interference among callback functions. It is the responsibility of a client to ensure compatibility among its callback functions and the callback functions of other clients.

Clients can also unregister a callback using the appropriate unregister routine (see **dr_events.h**). While unusual, it is possible for one callback routine to unregister another. In this case, DynamoRIO still calls routines that were registered before the event. Unregistration takes effect before the next event.

On Linux, an exec (SYS_execve) does NOT result in an exit event, but it WILL result in the client library

being reloaded and its **dr_client_main()** routine being called again. The system call events can be used for notification of SYS_execve.

Common Utilities

DynamoRIO provides clients with a powerful library of utilities for custom runtime code transformations. The interface includes explicit support for creating *transparent* clients. See the section on **Client Transparency** for a full discussion of the importance of remaining transparent when operating in the same process as the application. DynamoRIO provides common resources clients can use to avoid reliance on shared libraries that may be in use by the application. The client should only use external resources through DynamoRIO's own API, through DynamoRIO Extensions (see **DynamoRIO Extensions**), through direct system calls, or via an external agent in a separate process that communicates with the client (see **Communication**). Third-party libraries can be used if they are linked statically or loaded privately and there is no possibility of global resource conflicts (e.g., a third-party library's memory allocation must be wrapped): see **Using External Libraries** for more details.

DynamoRIO's API provides:

- Memory allocation: both thread-private (faster as it incurs no synchronization costs) and thread-shared
- Thread-local storage
- Thread-local stack separate from the application stack
- Simple mutexes
- File creation, reading, and writing
- Address space querying
- Application module iterator
- Processor feature identification
- Extra thread creation
- Symbol lookup (currently Windows-only)
- Auxiliary library loading

See **dr_tools.h** and **dr_proc.h** for specifics of each routine.

Another class of utilities provided by DynamoRIO are structures and routines for decoding, encoding, and manipulating IA-32, AMD64, and ARM instructions. These are described in **Instruction Representation**.

In addition, on Windows, DynamoRIO provides a number of utility functions that it fowards to a core Windows system library that we believe to be safe for clients to use:

- wcstoul
- wcstombs
- wcstol
- wcsstr
- wcsspn

- wcsrchr
- wcspbrk
- wcsncpy
- wcsncmp
- wcsncat
- wcslen
- wcscspn
- wcscpy
- wcscmp
- wcschr
- wcscat
- towupper
- towlower
- toupper
- tolower
- tan
- strtoul
- strtol
- strstr
- strspn
- strrchr
- strpbrk
- strncpy
- strncmp
- strncat
- strlen
- strcspn
- strcmp
- strchr
- sscanf
- sqrt
- sprintf
- sin
- qsort
- pow
- memset
- memmove
- memcpy
- memcmp
- memchr
- mbstowcs

- log
- labs
- isxdigit
- iswxdigit
- iswspace
- iswlower
- iswdigit
- iswctype
- iswalpha
- isupper
- isspace
- ispunct
- isprint
- islower
- isgraph
- isdigit
- iscntrl
- isalpha
- isalnum
- floor
- fabs
- cos
- ceil
- atol
- atoi
- atan
- abs
- _wtol
- _wtoi64
- _wtoi
- _wcsupr
- _wcsnicmp
- _wcslwr
- _wcsicmp
- _vsnprintf
- _ultow
- _ultoa
- _ui64toa
- _toupper
- _tolower
- _strupr

- _strnicmp
- _strlwr
- _stricmp
- _strcmpi
- _snwprintf
- _snprintf
- _memicmp
- _memccpy
- _ltow
- _ltoa
- _itow
- _itoa
- _i64tow
- _i64toa
- _ftol
- _fltused
- _chkstk
- _aullshr
- _aullrem
- _aulldiv
- _atoi64
- _allshr
- _allshl
- _allrem
- _allmul
- _alldiv
- __toascii
- __iscsymf
- __iscsym
- __isascii

In general, these routines match their standard C library counterparts. However, be warned that some of these may be more limited. In particular, _vsnprintf and _snprintf do not support floating-point values. DynamoRIO provides its own **dr_snprintf()** that does support floating-point values, but does not support printing wide characters. When printing floating-point values be sure to **save the application's floating point state** so as to avoid corrupting it.

64-Bit Reachability

To simplify reachability in a 64-bit address space, DynamoRIO guarantees that all of its code caches are located within a single 2GB memory region. It also places all client memory allocated through **dr_thread_alloc()**, **dr_global_alloc()**, **dr_nonheap_alloc()**, or **dr_custom_alloc()** with

DR_ALLOC_CACHE_REACHABLE in the same region.

DynamoRIO loads client libraries and Extensions (but not copies of system libraries) within 32-bit reachability of its code caches. Typically, the code cache region is located in the low 4GB of the address space; thus, to avoid relocations at client library load time, it is recommended to set a preferred client library base in the low 4GB.

The net result is that any static data or code in a client library, or any data allocated using DynamoRIO's API routines (except **dr_raw_mem_alloc()** or **dr_custom_alloc()**), is guaranteed to be directly reachable from code cache code. However, memory allocated through system libraries (including malloc, operator new, and HeapAlloc), as well as DynamoRIO's own internally-used heap memory, is *not* guaranteed to be reachable: only memory directly allocated via DynamoRIO's API. The **-reachable_heap runtime option** can be used to guarantee that all memory is reachable, at the risk of running out memory due to the smaller space of available memory.

To make more space available for the code caches when running larger applications, or for clients that use a lot of heap memory that is not directly referenced from the cache, we recommend that **dr_custom_alloc()** be called to obtain memory that is not guaranteed to be reachable from the code cache (by not passing **DR_ALLOC_CACHE_REACHABLE**). This frees up space in the reachable region.

When inserting calls, **dr_insert_call()** and **dr_insert_clean_call()** assume that the call is destined for encoding into the code cache-reachable memory region, when determining whether a direct or indirect call is needed. An indirect call will clobber r11. Use **dr_insert_clean_call_ex()** with **DR_CLEANCALL_INDIRECT** to ensure reachability when encoding to a location other than DR's regular code region, or when a clean call is not needed, **dr_insert_call_ex()** takes in a target encode location for more flexible determination of direct versus indirect.

DynamoRIO does not guarantee that any of its memory is allocated in the lower 4GB of the address space. However, it provides several features to make it easier to reference addresses absolutely:

- For directly referencing a global variable `var` in a client library, the client can create an operand with the address `&var` and it will auto-magically turn into a pc-relative addressing mode. OPND_CREATE_ABSMEM() directly creates a pc-relative operand, while **opnd_create_abs_addr()** will convert to a pc-relative operand when an absolute reference will not encode. An **opnd_create_rel_addr()** operand will also convert to an absolute reference when that will reach but a pc-relative reference will not.
- When using an address as an immediate, use the routines **instrlist_insert_mov_immed_ptrsz()** or **instrlist_insert_push_immed_ptrsz()** to conveniently insert either one or two instructions depending on whether the address is in the lower 4GB or not.
- When using an `instr_t` pointer as an immediate, use the routines **instrlist_insert_mov_instr_addr()** or **instrlist_insert_push_instr_addr()** to conveniently insert either one or two instructions depending on whether the resulting **instr_t** encoded address is in the lower 4GB or not.

String Encoding

All strings in the DynamoRIO API, whether input or output parameters, are encoded as UTF-8. DynamoRIO will internally convert to UTF-16 when interacting with the Windows kernel. A client can use **dr_snprintf()** or **dr_snwprintf()** with the `S` format code to convert between UTF-8 and UTF-16 on its own. (The _snprintf() function forwarded to ntdll does not perform that conversion.)

Building a Client

To use the DynamoRIO API, a client should include the main DynamoRIO header file:

```
#include "dr_api.h"
```

The client's target operating system and architecture must be specified by setting pre-processor defines before including the DynamoRIO header files. The appropriate library must then be linked with. The define choices are:

1. `WINDOWS`, `LINUX`, or (coming soon) `MACOS`
2. `X86_32`, `X86_64`, `ARM_32`, or (coming soon) `ARM_64`

Currently we provide a private loader for both Windows and Linux. With private loading, clients use a separate copy of each library from any copy used by the application.

If the private loader is deliberately disabled, for transparency reasons (see **Client Transparency**), clients should be self-contained and should not share libraries with the application. Without the private loader, 64-bit clients must take care to try and load themselves within reachable range of DynamoRIO's code caches by setting a preferred base address, although this may not always be honored by the system loader.

The DynamoRIO release supplies CMake configuration files to facilitate building clients with the proper compiler and linker flags. CMake is a cross-platform build system that generates Makefiles or other development system project files. A `DynamoRIOConfig.cmake` configuration file, along with supporting files, is distributed in the `cmake/` directory.

In its `CMakeLists.txt` file, a client should first invoke a `find_package(DynamoRIO)` command. This can optionally take a version parameter. This adds DynamoRIO as an imported target. If found, the client should then invoke the `configure_DynamoRIO_client()` function in order to configure build settings. Here is an example:

```
add_library(myclient SHARED myclient.c)
find_package(DynamoRIO)
if (NOT DynamoRIO_FOUND)
  message(FATAL_ERROR "DynamoRIO package required to build")
endif(NOT DynamoRIO_FOUND)
configure_DynamoRIO_client(myclient)
```

Note that when building a 32-bit client in Linux using `gcc`, the stack alignment should be 4-byte only. Using

the function `configure_DynamoRIO_client()` will configure the build settings correctly. Otherwise, appropriate options should be passed to the compiler: e.g., `-mpreferred-stack-boundary=2`.

The `samples/CMakeLists.txt` file in the release package serves as another example. The top of `DynamoRIOConfig.cmake` contains detailed instructions as well.

When configuring, the `DynamoRIO_DIR` CMake variable can be passed in to identify the directory that contains the `DynamoRIOConfig.cmake` file. For example:

```
mkdir ../build
cd ../build
cmake -DDynamoRIO_DIR=$DYNAMORIO_HOME/cmake ../myclient
make
```

The compiler needs to be configured prior to invoking cmake. If using gcc with a non-default target platform, the `CFLAGS` and `CXXFLAGS` environment variables should be set prior to invoking cmake. For example, to configure a 32-bit client when gcc's default is 64-bit:

```
mkdir ../build
cd ../build
CFLAGS=-m32 cmake -DDynamoRIO_DIR=$DYNAMORIO_HOME/cmake ../myclient
make
```

Note that `CXXFLAGS` should be set instead for a C++ client, and both should be set when building both types of clients from the same configuration (e.g., `samples/CMakeLists.txt`).

To improve clean call performance (see **Clean Calls** and **-opt_cleancall**), we recommend high levels of optimization when building a client.

If a client is not using CMake, the appropriate compiler and linker flags can be gleaned from `DynamoRIOConfig.cmake`. One method is to invoke CMake to generate a Makefile and then build with `VERBOSE=1`. We also summarize here the key flags required for 32-bit clients for `gcc`:

```
gcc -fPIC -shared -lgcc -DLINUX -DX86_32 -I$DYNAMORIO_HOME/include my-client.c
```

And for `cl`:

```
cl my-client.c /I$DYNAMORIO_HOME/include /GS- /DWINDOWS /DX86_32
   /link /libpath:$DYNAMORIO_HOME/bin dynamorio.lib /dll /out:my-client.dll
```

For a 64-bit client with `cl`:

```
cl my-client.c /I$DYNAMORIO_HOME/include /GS- /DWINDOWS /DX86_64
   /link /libpath:$DYNAMORIO_HOME/bin dynamorio.lib /dll /out:my-client.dll
   /base:0x72000000 /fixed
```

For 64-bit Linux clients, setting the preferred base takes several steps. Refer to `DynamoRIOConfig.cmake` for details.

To make clean call sequences more likely to be optimized, it is recommended to compile the client with optimizations, `-O2` for gcc or `/O2` for cl.

DynamoRIO Extensions

DynamoRIO supports extending the API presented to clients through separate libraries called DynamoRIO Extensions. Extensions are meant to include features that may be too costly to make available by default or features contributed by third parties whose licensing requires using a separate library. Extensions can be either static libraries linked with clients at build time or dynamic libraries loaded at runtime. A private loader is used to load dynamic Extensions.

Current Extensions provide symbol access and container data structures. Each Extension has its own documentation and has its functions and data structures documented separately from the main API. See the full list of Extensions here: **DynamoRIO Extensions**.

Be aware that some of the DynamoRIO Extensions have LGPL licenses instead of the BSD license of the rest of DynamoRIO. Such Extensions are built as shared libraries, have their own license.txt files, and clearly identify their license in their documentation. (We also provide static versions of such libraries, but take care in using them that their LGPL licenses match your requirements.)

Using External Libraries

Clients are free to use external libraries as long as those libraries do not use any global user-mode resources that would interfere with the running application, and as long as no alertable system calls are invoked on Windows (see **Avoid Alertable System Calls**). While most non-graphical non-alertable Windows API routines are supported, native threading libraries such as `libpthread.so` on Linux are known to cause problems.

Currently we provide a private loader for both Windows and Linux. Clients must either link statically to all libraries or load them using our private loader, which will happen automatically for shared libraries loaded in a typical manner. With private loading, the client uses a separate copy of each library from any copy used by the application. This helps to prevent re-entrancy problems (see **Resource Usage Conflicts**). Even with this separation, if these libraries use global resources there can still be conflicts. Our private loader redirects heap allocation in the main process heap to instead use DynamoRIO's internal heap. The loader also attempts to isolate other global resource usage and global callbacks. Please file reports on any transparency problems observed when using the private loader.

By default, all Windows clients link with libc. To instead use the libc subset of routines forwarded from the DynamoRIO library to `ntdll.dll` (which keeps clients more lightweight and is usually sufficient for most C code), set this variable prior to invoking configure_DynamoRIO_client():

```
set(DynamoRIO_USE_LIBC OFF)
```

C++ clients and standalone clients link with libc by default.

Avoid Alertable System Calls

On Windows, DynamoRIO does not support a client (or a library used by a client) making alertable system calls. These are system calls that can be interrupted for delivery of callbacks or asynchronous procedure calls. At the Windows API layer, they include many graphical routines, any Wait function invoked with `alertable=TRUE` (e.g., WaitForSingleObjectEx or WaitForMultipleObjectsEx), any Windows message queue function (GetMessage, SendMessage, ReplyMessage), and asynchronous i/o. In general, avoiding graphical, windowing, or asynchronous i/o library or system calls is advisable. DynamoRIO does not guarantee correct execution when a callback arrives during client code execution.

DynamoRIO Library Search Paths

DynamoRIO's loader searches for libraries in approximately the same manner as the system loader. It also has support for automatically locating Extension libraries that are packaged in the usual place in the DynamoRIO file hierarchy.

DynamoRIO supports setting DT_RPATH for ELF clients, via setting the DynamoRIO_RPATH variable to ON prior to invoking configure_DynamoRIO_client(). On Windows, setting that variable will create a "<client_basename>.drpath" text file that contains a list of paths. At runtime, DynamoRIO's loader will parse this file and add each newline-separated path to its list of search paths. This file is honored on Linux as well, though it is not automatically created there. This allows clients a cross-platform mechanism to use third-party libraries in locations of their choosing.

Deliberately Invoking Application Routines

Sometimes, a client wishes to invoke system library routines with the application context, rather than having them redirected and isolated by DynamoRIO. This can be accomplished using dynamic binding rather than static: dynamically looking up each desired library routine via DR's own routines (such as **dr_get_proc_address()**). (Using `GetProcAddress` will not work for this purpose as the result will be redirected.)

When Private Loader is Disabled

On Linux, if the private loader is deliberately disabled, ld provides the -wrap option, which allows us to override the C library's memory heap allocation routines with our own. For convenience, DynamoRIO exports **__wrap_malloc()**, **__wrap_realloc()**, and **__wrap_free()** for this purpose. These routines behave like their C library counterparts, but operate on DynamoRIO's global memory pool. Use the -Xlinker flag with gcc to replace the libc routines with DynamoRIO's _wrap routines, e.g.,

```
gcc -Xlinker -wrap=malloc -Xlinker -wrap=realloc -Xlinker -wrap=free ...
```

The ability to override the memory allocation routines makes it convenient to develop C++ clients that use the *new* and *delete* operators (as long as those operators are implemented using malloc and free). In particular, heap allocation is required to use the C++ Standard Template Library containers. When

developing a C++ client, we recommend linking statically to the C++ runtime library if not using the provided private loader.

On Linux, this is most easily accomplished by specifying the path to the static version of the library on the gcc command line. gcc's -print-file-name option is useful for discovering this path, e.g.,

```
g++ -print-file-name=libstdc++.a
```

A full gcc command line for building a C++ client when disabling the private loader (which is not the default) might look something like this (note that this requires static versions of the standard libraries that were built PIC, which is not the case in modern binary distributions and often requires building from source):

```
g++ -o my-client.so -I<header dir> \
  -fPIC -shared -nodefaultlibs \
  -Xlinker -wrap=malloc -Xlinker -wrap=realloc -Xlinker -wrap=free \
  `g++ -print-file-name=libstdc++.a` \
  `g++ -print-file-name=libgcc.a` \
  `g++ -print-file-name=libgcc_eh.a` \
  my-client.cpp
```

C++ Clients

The 3.0 version of DynamoRIO added experimental full support for C++ clients using the STL and other libraries.

On Windows, when using the Microsoft Visual C++ compiler, we recommend using the `/MT` compiler flag to request a static C library. The client will still use the `kernel32.dll` library but our private loader will load a separate copy of that library and redirect heap allocation automatically. Our private loader does not yet support locating SxS (side-by-side) libraries, so using `/MD` will most likely not work unless using a version of the Visual Studio compiler other than 2005 or 2008.

We do not recommend that a client or its libraries invoke their own system calls as this bypasses DynamoRIO's monitoring of changes to the process address space and changes to threads or control flow. Such system calls will also not work properly on Linux when using sysenter on some systems. If you see an assert to that effect in debug build on Linux, try the **-sysenter_is_int80** option.

Communication

Due to transparency limitations (see **Client Transparency**), DynamoRIO can only support certain communication channels in and out of the target application process. These include:

- DynamoRIO deployment control and runtime options: see **Deployment** and **Fine-Tuning DynamoRIO: Runtime Parameters**. In particular, the deployment API allows users to pass up-front runtime information to the client.
- Nudges: Since polling requires extra threads, and DynamoRIO tries not to create permanent extra threads (see **Thread Transparency**), a mechanism called *nudges* are the preferred mechanism for

pushing data into the process. Nudges are used to notify DynamoRIO that it needs to re-read its options, or perform some other action. DynamoRIO also provides a custom nudge event that can be used by clients. See **dr_nudge_process()** and **dr_register_nudge_event()**.
- Files can be used to send data out. An external process can wait on the file.
- Shared memory can be used for bi-directional communication. For an example of this on Windows, see the stats sample (see **Use of Custom Client Statistics with the Windows GUI**).

Annotations

DynamoRIO provides a binary annotation mechanism which allows the target application to communicate directly with the DynamoRIO client, or with DynamoRIO itself. A binary annotation is generated from a macro that can be manually inserted into the source code of the target program. When compiled, the resulting sequence of assembly instructions has no effect on native execution (i.e., it is a nop, or resolves to a static default value), but during execution under DynamoRIO, each annotation is detected and transformed into a function call to a set of registered handlers. Currently DynamoRIO provides 2 simple annotations:

- **DYNAMORIO_ANNOTATE_RUNNING_ON_DYNAMORIO()** Indicates by its return value whether the target app is running under DynamoRIO,
- **DYNAMORIO_ANNOTATE_LOG(format, ...)** Writes a message to the DynamoRIO log, when the target app is running under DynamoRIO and logging is enabled.

An annotation may be declared void, as in DYNAMORIO_ANNOTATE_LOG(), or it may have a return value, as in the boolean DYNAMORIO_ANNOTATE_RUNNING_ON_DYNAMORIO(). The return value can be used in a branch predicate, such that some of the target app's code only executes under DynamoRIO, or it can be used for in-process communication to obtain data from DynamoRIO or its client that is only available during binary translation.

Annotating an Application

Adding an annotation to a target application is primarily a simple matter of invoking the annotation macro at the desired program location. The macros are declared in the C header file **include/annotations/dr_annotations.h**, and each macro operates syntactically like a function call. An annotation having a return value can be used as an expression. DynamoRIO provides a module which defines the annotations, and clients may also provide modules containing custom annotations. Each compilation unit that uses annotations must be statically linked with the corresponding annotation module(s). For projects using cmake, a convenience function **use_DynamoRIO_annotations(target, srcs)** will configure the specified **srcs** to be linked with the annotation module.

Instrumenting Annotations

When DynamoRIO encounters an annotation in the target app, it instruments that program location in one of two ways:

- Return value substitution: DynamoRIO replaces the annotation with a constant return value (this instrumentation is only valid for annotations having a return value),
- Handler invocation: DynamoRIO replaces the annotation with a call to each handler that is currently registered for the annotation. For annotations having a return value, the return value of the last registered handler will be the value to take effect at the target program location.

Annotation handlers are registered using API functions **dr_annotation_register_call()** and **dr_annotation_register_return()**. Note that changes to handler registration will have no effect on annotations that have already been translated by DynamoRIO into the code cache (until the annotated basic blocks are removed from the cache and retranslated). The annotation instrumentation invokes the handlers using a separate clean call for each handler. The return value for an annotation can be set within a handler function using the API function **dr_annotation_set_return_value()**.

Creating Custom Annotations

DynamoRIO client developers may wish to create new annotations to facilitate client-specific communication with the target application. For example, a client that inspects memory usage may have false positives for variables in the target app that are never referenced after initialization. The **create_annotation** walks through the process of creating a new annotation that allows target application developers to explicitly mark any variable as defined.

Fine-Tuning DynamoRIO: Runtime Parameters

DynamoRIO's behavior can be fine-tuned using runtime parameters. Options are specified via `drconfig`, `drrun`, or **dr_register_process()**. See **Deployment**.

- **-no_follow_children**: By default, DynamoRIO follows all child processes. When this option is disabled via `-no_follow_children`, DynamoRIO follows only into child processes for which a configuration file exists (typically created by `drconfig`; see **Deployment**). On Linux, forked children are always followed and this option only affects execve.

  To follow all children in general but exclude certain children, leave `-follow_children` on (the default) and create config files that exclude the desired applications by running `drconfig` with the `-norun` option.

- **-opt_memory**: Reduce memory usage, but potentially at the cost of performance. This option can result in memory savings as high as 20%, and usually incurs no noticable performance degradation. However, it conflicts with the **-enable_full_api option** and cannot be used with **dr_unlink_flush_region()**.
- **-opt_cleancall** <*number*>: Optimize (shrink or inline) the clean call sequences (see **Clean Calls**). When DynamoRIO analyzes the callee and optimizes each clean call invocation, it assumes that a client will not modify the clean call callee or application instructions after the inserted clean call. If a client changes application instructions after an inserted clean call, the client may need to reduce the

-opt_cleancall level to preserve correct execution. The clean call will only be optimized if it is a leaf function. Currently, the callee will be inlined only if it is small, has at most one argument, and has no control flow other than for the PIC base. Compiling the client with optimizations makes clean call sequences more likely to be optimized. The optimization results (e.g. whether the inserted clean call is inlined or not, and which registers were saved on each context switch) are logged. Users can run DynamoRIO debug build with the runtime option "-loglevel 2 -logmask 0x02000000" (the logmask is optional but reduces the logfile size significantly) and grep for "CLEANCALL" in the log file to retrieve the information about clean call optimization. There are four optimization levels. By default, the clean call optimization level is 2.
    - 0: no optimization.
    - 1: callee register usage analysis and optimization on context switch.
    - 2: simple callee inline optimization, callee-save register analysis, and aflags usage analysis on the instruction list to be inserted.
    - 3: more aggressive, but potentially unsafe, optimizations.
- **-opt_speed**: By default, DynamoRIO provides a more straightforward code stream to clients in lieu of performance optimizations. This option attempts to obtain higher performance with potential loss of client simplicity. In particular, unconditional branches (both jumps and calls) and in some cases indirect calls may be elided in basic blocks. See also **Performance Limitations**. Note that **dr_insert_mbr_instrumentation()** is not supported when -opt_speed is specified.
- **-stack_size** <*number*>: DynamoRIO's per-thread stack is limited to 56KB by default (this may seem small, but this is much larger than its size when no client is present). This parameter can be used to increase the size; however, larger stack sizes use significantly more memory when targeting applications with hundreds of threads. The parameter can take a 'K' suffix, and must be a multiple of the page size (4K). This stack is used by the routines **dr_insert_clean_call()**, **dr_swap_to_clean_stack()**, **dr_prepare_for_call()**, **dr_insert_call_instrumentation()**, **dr_insert_mbr_instrumentation()**, **dr_insert_cbr_instrumentation()**, and **dr_insert_ubr_instrumentation()**. The stack is started fresh for each use, so *no persistent state may be stored on it*.
- **-thread_private**: By default, DynamoRIO's code caches are shared across threads. This option requests code caches that are private to each thread. For applications with many threads, thread-private code caches use more memory. However, they can be more efficient, particularly when inserting thread-specific instrumentation.
- **-disable_traces**: By default, DynamoRIO builds both a *basic block* code cache and a *trace* code cache (see **Instruction Representation**). This option disables trace building, which can have a negative performance impact. When running large, short-running applications, however, disabling traces can improve performance. When traces are disabled, **dr_register_trace_event()** has no effect. DynamoRIO tries to keep traces transparent to a client who is interested in all code and not only hot code, so there is rarely a reason to disable traces.
- **-enable_full_api**: DynamoRIO's default internal options balance performance with API usability. A few API functions, such as **dr_unlink_flush_region()**, are incompatible with this default mode. Client users can gain access to the entire set of API functions with -enable_full_api. Note that this

option may result in a small performance degradation.

- **-reachable_heap**: By default, DynamoRIO guarantees that heap allocated directly through its API routines **dr_thread_alloc()**, **dr_global_alloc()**, **dr_nonheap_alloc()**, or **dr_custom_alloc()** with **DR_ALLOC_CACHE_REACHABLE** is reachable by a 32-bit displacement from the code cache. However, it does not guarantee that memory allocated through system libraries (including malloc, operator new, and HeapAlloc) or DynamoRIO's own internal memory is reachable. Turning this option on combines all of the heap memory such that it is all guaranteed to be reachable from the code cache, at the risk of running out memory due to the smaller space of available memory.
- **-max_bb_instrs**: DynamoRIO stops building a basic block if it hits this application instruction count limit before hitting control flow or other block termination conditions. The default value is 1024; lower it if extensive client instrumentation is running into code cache size limit asserts.
- **-max_trace_bbs**: DynamoRIO will not build a trace with larger than this number of constituent basic block. The default value is 128; lower it if extensive client instrumentation is running into code cache size limit asserts.
- **-sysenter_is_int80**: This option only applies to Linux. If sysenter is the system call gateway, DynamoRIO normally hooks the vsyscall vdso page when it can. This option requests that DynamoRIO convert sysenter into int 0x80 instead. See **Using External Libraries**.
- **-multi_thread_exit**: By default, DynamoRIO synchronizes with all remaining threads at process exit time and the process exit event executes with only one live thread. This option requests that in release build the synchronization be avoided. The process exit event must be written in a thread-safe manner. Note that if thread exit events are registered, to avoid the synchronization the -skip_thread_exit_at_exit option must also be set. These options can also be enabled programmatically via **dr_set_process_exit_behavior()**.
- **-skip_thread_exit_at_exit**: By default, DynamoRIO synchronizes with all remaining threads at process exit time in order to safely call each thread exit event. This option requests that in release build the synchronization be avoided by removing the invocation of thread exit events at process exit time. Note that if the process exit event is registered, to avoid the synchronization the -multi_thread_exit option must also be set. These options can also be enabled programmatically via **dr_set_process_exit_behavior()**.
- **-persist**: Enables persisting of code caches to disk and re-use on subsequent runs. Caches are persisted in units that correspond to application libraries, or sometimes smaller units. Each unit is persisted to its own file in a subdirectory of the base directory specified by `-persist_dir`. See **Persisting Code** for more details.
- **-persist_dir** <*path*>: Sets the base directory for persistent code cache files. If unset, the default base directory is the log directory. A different sub-directory will be created for each user inside the specified directory.
- **-translate_fpu_pc**: Enables translation of the last floating-point instruction address when the last floating-point instruction is not in the same basic block as the instruction saving the FPU state. This is off by default as it incurs significant performance penalties and few applications require this feature.
- **-syntax_intel**: This option causes DynamoRIO to output all disassembly using Intel syntax rather

than the default show-implicit-operands syntax. This can also be set using **disassemble_set_syntax()**.

- **-syntax_att**: This option causes DynamoRIO to output all disassembly using AT&T syntax rather than the default show-implicit-operands syntax. This can also be set using **disassemble_set_syntax()**.
- **-syntax_arm**: This option causes DynamoRIO to output all disassembly using standard ARM assembler syntax rather than the default show-implicit-operands syntax. This can also be set using **disassemble_set_syntax()**.
- **-disasm_mask**: This option sets the disassembly style to the specified bitmask of dr_disasm_flags_t values. This option overlaps with -syntax_intel, -syntax_att, and -syntax_arm. The style can also be set using **disassemble_set_syntax()**.
- **-tracedump_text** and **-tracedump_binary**: These options cause DynamoRIO to output all traces that were created to the log file *traces-shared.0.TID.html*, where *TID* is the thread id of the initial thread; any thread-private traces (see **-thread_private option**) produce per-thread files *traces.TID.html*. Traces are logged whenever they are flushed from the cache (which can be during execution or at the latest at program termination). The two options select either a text dump or a binary dump. The text dump takes up considerable room and time to dump, while the binary dump requires more effort to examine. The binary trace dump format is documented in **dr_tools.h**, and a sample reader is provided with this distribution.
- **-tracedump_origins** When selected by itself with neither -tracedump_text nor -tracedump_binary, dumps only a text list of the constituent basic block tags of each trace to the trace log file. When combined with either of -tracedump_text or -tracedump_binary, adds a full disassembly of the constituent basic blocks to the selected dump.

Options controlling notifications from DynamoRIO:

- **-msgbox_mask** *0xN*: Controls whether DynamoRIO uses pop-up message boxes on Windows, or waits for a key press on Linux, when presenting information. The mask takes the following bitfields:
  - INFORMATION = 0x1
  - WARNING = 0x2
  - ERROR = 0x4
  - CRITICAL = 0x8

  dr_messagebox() is not affected by -msgbox_mask. For the provided Windows debug build -msgbox_mask defaults to 0xC. On Linux the default is 0, as this feature reads from standard input and might conflict with some applications. On Linux the pause can be changed to use an infinite loop rather than reading from standard input by passing the **-pause_via_loop** runtime option, which allows attaching a debugger.

Attention

On Vista or higher most Windows services are currently unable to display message boxes (see **Limitations**). Since these services also don't have an associated console for stderr printing, the **-loglevel** and **-logmask** options should be used instead. For the messages that would be displayed

by -msgbox_mask, setting any bit in -logmask is sufficient for the message to be included in the logfile.

- **-stderr_mask** *0xN*: Parallel to -msgbox_mask, but controls DynamoRIO's output to standard error. This option takes the same bitfields as -msgbox_mask. The API routine **dr_is_notify_on()** can be used to determine if -stderr_mask is non-zero. Messages printed to stderr will only be visible for applications that have an attached console. They will not be visible in the `cmd` console on Windows 7 or earlier or on any Windows version when running a graphical application in `cmd` (even with **dr_enable_console_printing()**, as that only affects clients calling **dr_printf()** or **dr_fprintf()**) but the output can be viewed from `cmd` by redirecting to a file. For the provided Linux debug builds, -stderr_mask defaults to 0xF; for the Linux release builds, its default is 0xE. The default on Windows is 0.

Options aiding in debugging:

- **-no_hide**: By default, DynamoRIO hides itself from the Windows module list, for transparency. However, this makes it more difficult to debug a process under DynamoRIO's control. The option -no_hide turns off this module hiding. However, the client library and any libraries it imports from will still be hidden. We provide a windbg script that can locate DynamoRIO, the client library, and all of its dependences, so this option should no longer be necessary (see **Diagnosing and Reporting Problems**). This option is for Windows only.

Options available only in the debug build of DynamoRIO:

- **-loglevel** *N:* If N is greater than 0, DynamoRIO prints out a log of its actions. The greater the value of N, the more information DynamoRIO prints. Useful ranges are from 1 to 6. Verbosity is set to 0 by default, i.e., no log written. All log files are kept in a log directory. There is one directory per address space per run. The directories are named *app.NNN*, where *app* is the application name and *NNN* is a number that is incremented with each directory created. On Windows the directories are located by default in a subdirectory *logs* of the DynamoRIO home directory as specified in the **dr_register_process()**, `drconfig`, or `drrun` configuration for the target application. The runtime option **-logdir** can be used to override the default directory. There is one main log file per directory named *app.0.TID.html*, where *TID* is the thread identifier of the initial thread. There is also a log file per thread, named *log.N.TID.html*, where *N* is the thread's creation ordinal and *TID* is its thread identifier. The loglevel may be changed during program execution, but if it began at 0 then it cannot be raised later. The -logmask parameter can be used to control which DynamoRIO modules output data to the log files. **dr_log()** allows the client to write to the above logfiles.

- **-logmask** *0xN*: Selects which DynamoRIO modules print out logging information, at the -loglevel level. The mask is a combination of the LOG_ bitfields listed in **dr_tools.h** (**LOG_ALL** selects all modules).

- **-logdir** <*path*>: Specifies the directory to use for log files. See the documentation for **-loglevel** for a description of the default log directory.

- **-ignore_assert_list '*'**: Ignores all DynamoRIO asserts of the form "<file>:1234". * may be replaced by a ; separated list of individual asserts to ignore "foo.c:333;bar.c:12".

Diagnosing and Reporting Problems

When using a complex system like DynamoRIO, problems can be challenging to diagnose. This section contains some debugging tips and shows how to get help.

Obtaining Help and Reporting Problems

For questions and discussion, join the DynamoRIO Users group.

For bug reports, use the Issue Tracker. Please include a detailed description of the problem (is it an application crash? a DynamoRIO crash? a hang? a debug build assert?) and how to reproduce it.

Troubleshooting

- DynamoRIO disables itself when Windows is booted in safe mode (without networking). Thus, if a crash occurs in a Windows service under DynamoRIO, rebooting in safe mode will allow recovery.
- If the client library doesn't seem to function for a given process, it is likely that the client library wasn't loaded due to errors.

  One of the common situations where this happens is when the target application runs as a different user than the user who created the client library. This results in the application process not having the right permissions to access the client library.

  Try running the process under the debug mode of DynamoRIO (see **dr_register_process()**), where diagnostic messages are raised on errors like client library permissions. To see all messages, set the notification options like -msgbox_mask and -stderr_mask options to 0xf (see **Fine-Tuning DynamoRIO: Runtime Parameters**). This will alert you to the problem.

- DynamoRIO asserts of the form "<file>:1234" can be suppressed with the **-ignore_assert_list '*'** option. * may be replaced by a ; separated lists of individual asserts to suppress as so "-ignore_assert_list 'foo.c:333;bar.c:12'".
- The DynamoRIO header files have typedefs that may conflict with other header files wrapped in ifndef DR_DO_NOT_DEFINE_<type> to make it easier to work around such conflicts.

Using Debuggers

A process under control of DynamoRIO can be executed within a debugger. For debugging on Windows we recommend using windbg version 6.3.0017 (**not** the newer versions, as they have problems displaying callstacks involving DynamoRIO code).

Normally, the debugger will not be aware of the DynamoRIO library or the client library. We provide a

windbg script that locates the DynamoRIO library, the client library, and any privately-loaded dependent libraries. The script is in `bin32/load_syms.txt` and `bin64/load_syms64.txt`. To load it from windbg, execute the following command:

```
$><c:\path\to\DR\bin32\load_syms.txt
```

When debugging often, modify the shortcut that launches windbg to include this command as a -c argument. E.g.:

```
"C:\Program Files (x86)\Debugging Tools for Windows\windbg.exe" -pt 1 -c "$>
       <c:\tools\DynamoRIO\bin32\load_syms.txt"
```

On Windows, the **-no_hide** option can alternatively be used so the debugger can see the DynamoRIO library, but the debugger will still not be able to see the client library or any of its dependent libraries. We recommend using our script.

To attach to a process on Windows, use the **-msgbox_mask** option and attach the debugger while the dialog box has paused the application. On Linux, the same option can be used and the debugger attached while the application waits for enter to be pressed. Since this may not work for applications that themselves read from standard input, we also provide the **-pause_via_loop** runtime option which sits in an infinite loop rather than waiting for a keypress.

To run an application on Linux under a debugger from process start you can launch drrun under gdb as you would normally:

```
gdb --args path/to/drrun <options> -- path/to/app
```

Because the executable changes from drrun to the app, the app cannot be re-run from gdb's prompt.

On Linux, the main drawback of debugging from application start rather than attaching is that breakpoint instructions (`int3`) inserted by the debugger get copied into the code cache. This includes internal debugger breakpoints automatically placed in the loader, as well as user-defined breakpoints. For example, gdb puts a breakpoint on __nptl_create_event, which is called by pthread_create and related calls. See https://github.com/DynamoRIO/dynamorio/issues/490. The debugger will handle these traps, but the user must tell it to continue, which is an annoyance. For user breakpoints, consider using read watchpoints on the code in question instead.

On Windows, if an application invokes OutputDebugString() while under a debugger, DynamoRIO can end up losing control of the application.

For additional tips, check the DynamoRIO wiki page on debugging:
https://github.com/DynamoRIO/dynamorio/wiki/Debugging