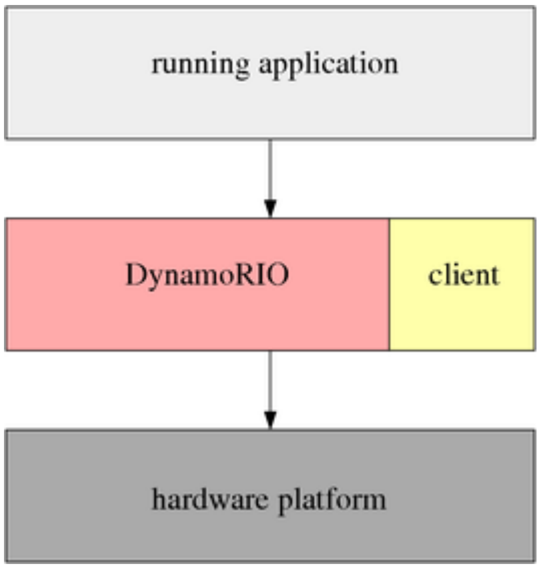
Usage Model for DynamoRIO

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

- **Deployment**
- **Common Events**
- **Common Utilities**
- **Building a Client**
- **Using External Libraries**
- **DynamoRIO Extensions**
- **Communication**
- **64-Bit Reachability**
- **Fine-Tuning DynamoRIO: Runtime Parameters**
- **Diagnosing and Reporting Problems**

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 Disassembly Library**).

Deployment

Once the DynamoRIO distribution contents are unpacked (see **Distribution Contents**), configuration and execution of applications under DynamoRIO is handled by a set of libraries and tools. On Windows, the tools are `drconfig.exe`, `drrun.exe`, and `drinject.exe`. The corresponding libraries (whose APIs are exposed by the tools) are `drconfiglib.dll` and `drinjectlib.dll` with header files **`dr_config.h`** and **`dr_inject.h`**. On Linux, the tools are the `drconfig`, `drrun`, and `drinject` scripts.

When using DynamoRIO as a third-party disassembly library (see **IA-32/AMD64 Disassembly Library**), no deployment is needed, as DynamoRIO does not control a target application when used as a regular library.

Windows Deployment

There are two methods for running a process under DynamoRIO: the one-time configure-and-run, and the two-step separate configuration and execution. The `drrun.exe` tool supports the first, simpler model, while the `drconfig.exe` and `drinject.exe` tools support the second, more powerful model. The `drconfig.exe` tool, or the corresponding the `drconfiglib.dll` library, can also be used to **nudge** running processes.

Configuration information is stored in files in the current user's profile directory, which is obtained from the environment variable `$USERPROFILE`. Thus, configurations are persistent across reboots and are private to each user. DynamoRIO also supports global configurations, which are stored in the "config" subdirectory of the directory specified by the `DYNAMORIO_HOME` registry value in the registry key `\HKLM\SOFTWARE\DynamoRIO\DynamoRIO` (or for 32-bit on 64-bit Windows (WOW64) `\HKLM\SOFTWARE\Wow6432Node\DynamoRIO\DynamoRIO`). Setting that `DYNAMORIO_HOME` value and creating the directory it points to must be done manually. The provided tools support reading and writing both local and global configuration files, and automatically creating the local directory. DynamoRIO gives local files precedence when both exist. Note that applications that do not have a $USEPROFILE environment variable can only be executed using global configurations.

Configurations are per-process, with the basename of the process used for identification (e.g., `calc.exe`). One-time configuration also uses the process id to specify that the configuration is for that process instance only.

As an example, assume you have unpacked the DynamoRIO distribution and your current directory is its base directory. Run `calc.exe` with the bbsize sample client using the following configure-and-run command:

```
bin32/drrun.exe -client samples/bin32/bbsize.dll 0 "" calc
```

To use system-wide injection, allowing for an application to be run under DynamoRIO regardless of how it is invoked, configure the application first (-syswide_on requires administrative privileges):

```
bin32/drconfig.exe -reg calc.exe -syswide_on -client samples/bin32/bbsize.dll 0 ""
```

The next time `calc.exe` is started by the current user, it will run under DynamoRIO with the bbsize client.

To unregister `calc.exe`, issue the following command:

```
bin32/drconfig.exe -unreg calc.exe
```

Invoke any of the `drconfig.exe`, `drrun.exe`, or `drinject.exe` tools with no arguments to see the full list of options available.

By default on Windows DynamoRIO only follows into children that are configured (via `drconfig.exe`). To follow all children, use the **-follow_children** runtime option.

To **nudge** all instances of `calc.exe` running under DynamoRIO with argument "5", use:

```
bin32/drconfig.exe -nudge calc.exe 0 5
```

This will result in a nudge event with argument=5 delivered to the client callback registered with **dr_register_nudge_event()** in all `calc.exe` processes running under DynamoRIO. The third argument, 0, is an ID supplied at registration which uniquely identifies the target client (see dr_deploy.h for details). Note that nudging 64-bit applications is not yet supported on Windows.

To view 32-bit or WOW64 processes running under DynamoRIO the `drview.exe` tool can be used. The bin64 version will display both 32-bit and 64-bit processes and will indicate which are 32-bit. The bin32 version will display 64-bit processes but is unable to determine whether DynamoRIO is present.

Attention:
Note that on Windows NT a reboot is required after using -syswide_on or -syswide_off.

DynamoRIO uses the `\HKLM\SOFTWARE\Microsoft\Windows\Windows NT\CurrentVersion\AppInit_DLLs` key (for 32-bit on 64-bit Windows (WOW64), `\HKLM\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs`) for -syswide_on to inject into new processes without having to directly launch them `drrun.exe` or `drinject.exe`. For injection to work, the registered process must statically link to user32.dll (only a few small non-graphical windows applications don't link user32.dll). If a target application does not link to user32.dll, DynamoRIO can still inject if the process is launched with `drinject.exe` or if the parent process (usually cmd.exe or explorer.exe for user launched processes) is running under DynamoRIO. The drinject.exe tool uses the configuration information set by `drconfig.exe` for the target application.

Attention:
The -syswide_on, -syswide_off, use of global configuration files, and nudging certain processes may require administrative privileges. On Windows Vista, if UAC is enabled, use an elevated (runas admin) process. When using `drconfig.exe` and `drrun.exe` in these scenarios, be sure that the cmd shell being used was started with elevated permissions.

Linux Deployment

Once DynamoRIO has been unpacked, a set of three scripts provide flexibility in configuring and executing applications. The **LD_PRELOAD** based scripts and the **app_start()/app_stop()** interface are currently the only supported methods of running applications under DynamoRIO on Linux.

There are two methods for invoking an application under DynamoRIO:

1. Configure and launch in one step via `drrun`
2. Configure via `drconfig` and launch via `drinject`

As an example of the simpler method, the following command runs `ls` under DynamoRIO with the bbsize sample client:

```
% bin32/drrun -client samples/bin32/libbbsize.so 0 "" ls
```

Run `drrun` with no options to get a list of the options and environment variable shortcuts it supports. To disable following across child execve calls, use the **-no_follow_children** runtime option.

Use the scripts in `bin32/` for 32-bit applications and the scripts in `bin64/` for 64-bit applications.

The two-step method allows for greater control over child processes. The `drconfig` script writes a configuration file for a given application name. DynamoRIO reads its options from the configuration file at runtime. Once each process name is configured, the `drinject` script can be used to invoke the parent process. The `drrun` script can also be used but it creates a temporary configuration file that will override settings requested via `drconfig`. The configuration file for each application is stored in `$HOME/.dynamorio/<appname>.config32` (or a `config64` suffix for 64-bit). DynamoRIO also supports global configuration files in `/etc/dynamorio/<appname>.config32` when a local configuration file is not found. `drconfig` does not support directly writing a global config file but such files can be copied from or modeled on local files.

If a target application executes an `execve` that discards the `$HOME` environment variable, the resulting process will not run under DynamoRIO control. Use global configuration files to handle this situation.

To **nudge** a process with pid `targetpid` running under DynamoRIO and pass argument "5" to the nudge callback, use the `nudgeunix` tool:

```
bin32/nudgeunix -pid targetpid -client 0 5
```

This will result in a nudge event with argument=5 delivered to the client callback registered with **dr_register_nudge_event()** in the target process. The 0 argument is an ID supplied at registration which uniquely identifies the target client (see dr_deploy.h for details).

Multiple Clients

DynamoRIO does support multiple clients. It is each client's responsibility, however, to ensure compatibility with other clients. DynamoRIO makes no attempt to force cooperation among clients. For example, instruction stream modifcations made by one client are visible to other clients. Systems employing multiple clients must be aware of such interactions and design accordingly.

Client registration requires users to specify the *priority* of each client. DynamoRIO calls each client's **dr_init()** routine sequentially according to this priority. Clients with a numerically lower priority value are called first and therefore given the first opportunity to register callbacks (the client with priority 0 is called first). Since DynamoRIO delivers event callbacks sequentially, client priority and the order of event registration is important. For a given event, the *first* registered callback is called *last*. This scheme gives precedence to the first registered callback since that callback is given the final opportunity to modify the instruction stream or influence DynamoRIO's operation.

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
- Process initialization and exit
- Thread initialization and exit
- Fork child initialization (Linux-only); meant to be used for re-initialization of data structures and creation of new log files
- Application library load and unload
- Application fault or exception (signal on Linux)
- System call interception: pre-system call, post-system call, and system call filtering by number
- Signal interception (Linux-only)
- Nudge received - see **Communication**

Typically, a client will register for the desired events at initialization in its **dr_init()** 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()**) 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_init()** 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_init()** 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)

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 and AMD64 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 and heap are within the same 2GB memory region. DynamoRIO also loads client libraries within 32-bit reachability of its code caches and heap (which currently requires that client libraries have preferred bases in the lower 2GB of the address space: see **limits_64bit**). This means that any static data or code in a client library, or any data allocated using DynamoRIO's API, is guaranteed to be directly reachable from code cache code.

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` or `LINUX`
2. `X86_32` or `X86_64`

For transparency reasons (see **Client Transparency**), clients should be self-contained and should not share libraries with the application. Thus, when a client is linked the linker options for no default libraries, no startup files, and no entry point should be used. Not specifying those options will result in an unsafe client library with the potential to deadlock or crash.

Additionally, 64-bit clients must set a preferred base address in the lower 2GB.

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)
```

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`).

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 -nostartfiles -nodefaultlibs -lgcc -DLINUX -DX86_32
    -I$DYNAMORIO_HOME/include my-client.c
```

And for `cl`:

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

For a 64-bit client with `cl`:

```
cl my-client.c /I$DYNAMORIO_HOME/include /GS- /DWINDOWS /DX86_64
   /link /NODEFAULTLIB /NOENTRY
   /libpath:$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.

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 on Windows; a private loader for Linux is in progress.

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.

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. Furthermore, clients must either link statically to all libraries or load them using a private loader separate from the application's loader in order to prevent re-entrancy problems (see **Resource Usage Conflicts**).

Currently we provide a private loader for Windows and we plan to supply a private Linux loader in the near future. With private loading, the client uses a separate copy of each library from any copy used by the application. Even with this separation, if these libraries use global resources there can still be conflicts. Our Windows 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. In this release it should be considered a beta feature. Please file reports on any transparency problems observed when using the private loader.

On Linux, where we do not yet have a private loader, 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 ...
```

C++ Clients

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.

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 might look something like this:

```
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
```

See also the stl_test.cpp sample and CMake build files provided with this documentation. The provided CMake build files will set these flags automatically when the `DynamoRIO_CXX` CMake variable is set (see **Building a Client**).

On Windows, when using the Microsoft Visual C++ compiler, simply use the `/MT` compiler flag. 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 libraries, so using `/MD` will most likely not work unless using an older version of the compiler.

Be aware that we have successfully built and tested several small C++ clients using the gcc toolchain, but have not performed extensive testing. Our clients successfully compile and run linked with the g++ libraries included in Red Hat Enterprise Linux 4, but on later distributions the pre-built libraries may not work and you may need to build static versions from the gcc sources with the appropriate flags (including -fPIC).

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.

Fine-Tuning DynamoRIO: Runtime Parameters

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

- **-follow_children**: This option only applies to Windows. By default, DynamoRIO follows into only child processes configured via `drconfig.exe`. When `-follow_children` is specified DynamoRIO injects into all child processes.

- **-no_follow_children**: This option only applies to Linux. By default, DynamoRIO follows into child processes across execve. When `-no_follow_children` is specified DynamoRIO only injects across an execve if a configuration file exists (typically created by `drconfig`: see **Deployment**) for the new application name.

- **-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()**.

- **-stack_size** <*number*>: DynamoRIO's per-thread stack is limited to 20KB 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 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.

- **-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**.

- **-synch_at_exit**: In debug builds, DynamoRIO synchronizes with all remaining threads at process exit time. In release build, for performance reasons, while DynamoRIO attempts to prevent other threads from executing (and thus running instrumentation code or raising events) beyond when their own thread exit events are raised, no guarantee is provided. If the client spends a long time in the process exit event, concurrent threads could cause racy access to data and thus errors. This option can be turned on in order to provide a guarantee and avoid races, but at a potential performance hit. This option can also be enabled programmatically via **dr_request_synchronized_exit()**.

- **-syntax_intel**: This option causes DynamoRIO to output all disassembly using Intel syntax rather than the default AT&T-style 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.

Attention:
On Vista 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. 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:

- **-pause_on_error**: For Linux builds only, this option requests that when DynamoRIO encounters an assert or crash that it suspend the process so that a debugger can be attached.
- **-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. 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. The directories are located by default in a subdirectory *logs* of the DynamoRIO home directory as specified in the **dr_register_process()** or `drconfig.exe` configuration for the target application. On Linux the `drconfig` and `drrun` scripts allow for setting the logging 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).
- **-ignore_assert_list '*'**: Ignores all DynamoRIO asserts of the form "<file>:1234". * may be replaced by a ; separated lists 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.

Debugging Tips

- 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'".
- A process under control of DynamoRIO can be executed within a debugger. On Windows it is best to set the **-no_hide** option so the debugger can see the DynamoRIO library.

  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, an alternative exists: the **-pause_on_error** option allows attaching a debugger when a problem occurs. To run an application on Linux under a debugger from the start you can perform the work of the `drrun` script by setting the LD_LIBRARY_PATH and LD_PRELOAD environment variables from within the debugger.

  For debugging on Windows we recommend using windbg version 6.3.0017 (**not** the newer versions 6.4 through 6.8, as they have problems displaying callstacks involving DynamoRIO code).

  Take care when setting breakpoints in an application running under DynamoRIO. If the debugger inserts `int3` it can find its way into the code cache and cause errors as the debugger will not realize that the resulting trap is from a breakpoint. Use read watchpoints on the code in question instead.

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