A Progress Report of the Manchester University ARM Port of Dynamo-RIO

James Sandford
School of Computer Science,
The University of Manchester,
Oxford Road,
Manchester,
M13 9PL
sandfoj9@cs.man.ac.uk

September 9, 2011

Abstract

This report has been put together following several weeks of trying to understand the Dynamo-RIO software project and the current progress made in it's port to the ARM architecture from the original x86 code carried out originally by Stephen Barton as a final year computer science project in the year prior to this report. I aim to provide a means to get to my current position as quickly as possible by showing how to get the code cross compiling on an x86 Linux system, pitfalls I fell down when going through this process and others, an overview of the code structure and an overview of what has and has not been completed in the port at this point in time. In addition to this, I will provide any advice I can on working with the current code base and I will provide recommendations on how to progress from here.

Contents

1 The code base

1.1 Git and GitHub

The code for the ARM port is currently available on GitHub and can be found at `http://github.com/j616/DynamoRIO-for-ARM/`. I shalln't go into any detail on how to use Git or GitHub here but you can find all you need to know at `http://help.github.com/`. But Git is basically a version control system that's a bit nicer when it comes to things such as merging than systems like SVN. GitHub is just a hosting website for Git repositories that provides a nice web interface with some occasionally useful features such as graphing of contribution and programming languages along with the usual browser based viewing and editing of files.

1.2 The Folder Structure

The folder structure consists of the following:

Clients Several example clients.

CurrentVersion The copy of the source being worked on. This is our main focus.

- **api** Strangely named as it contains the standard Dynamo-RIO sample clients such as bbcount.
- **bin** The shell scripts to run Dynamo-RIO. For information on these, look at the README in the CurrentVersion folder.
- **clients** A couple of clients. These are Windows specific so I haven't looked at them. I should note here that the ARM port is currently Linux specific as Windows doesn't currently support ARM. Though I believe this will change with Windows 8.
- **cmake** CMake configuration files. Don't really need touching but hold things like the version number to give your build.
- **core** The place where things get interesting. This is Dynamo-RIO itself in many ways. This is the code that manages everything else. Importantly, this is where we maintain control of the software we are working on along with the decoding, modification and encoding of instructions. Most (all?) of the ARM specific code will be in here.
  - **Arm** The (mainly) ARM specific files. More on this later.
  - **lib** The header files and generation scripts for the client libraries.
  - **linux** The Linux specific files.
  - **win32** The Windows specific files (not just 32bit).
  - **x86** The (mainly) x86 specific files. More on this later.
- **ext** Extensions. I believe this is needed for the actual use of clients in Dynamo-RIO.
- **include** The basic header files you will use when constructing clients. There is a major problem here that will be discussed later.
- **libutil** General libraries. I haven't needed to touch these.

- **make** Auto-generated make files.
- **suite** A suite of development tools for the main Dynamo-RIO devs. This hasn't and probably shouldn't be used in the ARM build at least while it remains a separate project.
- **tools** Again, a large number of development tools. These again look mainly related to the main version of Dynamo-RIO and to x86. Some may be adaptable though for debugging and testing.

Decoder A standalone ARM machine code decoder. This will be explained more later.

Encoder A standalone ARM machine code encoder. This too will be explained a little more later.

MiscCode Samples of ARM assembly, x86 assembly, very simple test C programs for running Dynamo-RIO on and CMake script examples.

- **ArmAssembly** Some small test programs in C and ARM assembly. More on this later.
- **Assembly** Some small test programs in x86 assembly.
- **cMakeExamples** Some small example scripts for CMake.

RunningVersion A copy of the source containing mainly x86 specific code. This has remained largely un-used by myself and thus I won't go into any detail. Best sticking with CurrentVersion.

1.3 Notation

My predecessor employed the following notation to keep track of his work:

COMPLETEDD This function has been finished and fully tested. Note the capitals and the extra 'D'.

INPROCESSS This function is currently being worked on and has not been fully tested. Note the capitals and the extra 'S'.

ADDME Major functionality needs adding or adapting here. Note the capitals.

In addition to this, there is a healthy spread of 'fixme' comments. At least some of these are from the main Dynamo-RIO project though some may be from the ARM port.

1.3.1 INPROCESSS Tree

Below is a tree I have constructed of the INPROCESSS functions and their connected usage. This is based on a simple grep of the INPROCESSS functions and is by no means a complete list of what needs doing.

```
core/linux/preload.c:_init(
 core/dynamo.c:Dynamorio_app_init(
  core/dynamo.c:dynamo_thread_init(
   core/dynamo.c:is_thread_initialized(
   )
   core/fcache.c:fcache_reset_all_caches_proactivly(
    core/synch.c:synch_with_all_threads(
     core/synch.c:synch_with_thread(
      core/dynamo.c:dynamo_other_thread_exit(
       core/dynamo.c:dynamo_thread_exit_common(
        core/fragment.c:enter_threadexit(
         core/fragment.c:check_flush_queue(
          core/vmareas.c:vm_area_flush_fragments(
           core/fragment.c:fragment_delete(
            core/fragment.c:fragment_output(
            )
            core/monitor.c:monitor_remove_fragment(
             core/monitor.c:trace_abort(
              core/monitor.c:internal_restore_last(
               core/link.c:link_fragment_outgoing(
                core/link.c:is_linkable(
                 core/monitor.c:monitor_is_linkable(
                  core/monitor.c:should_be_trace_head(
                   core/monitor.c:should_be_trace_head_internal(
                   )
                  )
                 )
                )
               )
              )
             )
            )
           )
          )
         )
        )
       )
      )
     )
    )
   )
  )
 )
)
core/link.c:unlink_fragment_incoming(
 core/link.c:unlink_branch(
 )
)
core/linux/os.c:dr_syscall_invoke_another(
)
core/linux/signal.c:thread_set_self_context(
)
core/utils.h:FATAL_USAGE_ERROR(
)
core/configc.c:config_reread(
)
core/Arm/emit_utils.c:emit_fcache_enter_shared(
 core/Arm/emit_utils.c:emit_fcache_enter_common(
 )
)
core/Arm/emit_utils.c:coarse_exit_prefix_size(
)
core/Arm/emit_utils.c:patch_branch(
)
core/Arm/emit_utils.c:coarse_is_entrance_stub(
)
core/Arm/emit_utils.c:emit_fcache_enter(
 core/Arm/emit_utils.c:emit_fcache_enter_common(
 )
)
core/Arm/instrument.c:instrument_init(
)
core/Arm/decode.c:decode_operand(
)
core/Arm/mangle.c:remangle_short_rewrite(
)
core/Arm/mangle.c:sandbox_top_of_bb(
)
core/Arm/arch.c:arch_thread_init(
 core/Arm/mangle.c:set_selfmod_sandbox_offsets(
 )
)
core/Arm/instr.c:instr_get_branch_target_pc(
)
core/Arm/instr.c:instr_is_syscall(
)
core/Arm/instr.c:instr_is_cti_short_rewrite(
)
core/Arm/arch.c:shared_gencode_init(
 core/Arm/emit_utils.c:emit_fcache_enter_shared(
 ****SEE ABOVE TRACE****
 )
)
core/Arm/arch.c:recreate_app_state(
 core/Arm/arch.c:recreate_app_state_internal(
 )
)
core/Arm/disassemble.c:instr_disassemble(
)
core/Arm/disassemble.c:dump_dr_callstack(
)
core/fragment.c:update_lookable_tls(
)
```

```
core/fragment.c:fragment_recreate_with_linkstubs(
)
core/options.c:synchronize_dynamic_options(
)
```

2 Working With the Code

2.1 Platform Specific Settings and Tools

2.1.1 Native ARM Compilation

While it should be theoretically possible to compile natively on an ARM device, problems are present in that assembly takes quite a large amount of memory. So when trying to compile on the Tegra 250, I found memory usage goes up and up until it gives up an the process dies. This would probably be solved by having swap space. If you want to try, run CMake with the ARM option on and the cross-compile option off. The difference with cross-compile being the build commands used. The advantage of this is you can build on the machine under test and thus use specific Linux kernel modules rather than generic ones. As stated earlier, ARM support is only for Linux.

2.1.2 Cross-Compiling from x86

The main tool you'll need to cross-compile is the CodeSourcery Lite toolchain. You can get this from `http://www.codesourcery.com/sgpp/lite/arm/portal/subscription?@template=lite`. You want the GNU/Linux version as this contains all the generic cross-compiled headers required by Dynamo-RIO. Once you've got this, install it with the nice GUI provided and all should be blue skies and roses with any luck. You want to run CMake with both the ARM and cross-compile options on.

2.2 Compilation

A couple of more notes on CMake settings. To get to the options, use the `cmake -i` command. This gives you an interactive prompt. Running `cmake` on it's own will create make scripts using the last settings used. You shouldn't need the advanced options. One big note is that you won't be able to build the extensions library or samples. These require the api to build but this is not yet possible due to missing functions.

Once you have changed any CMake settings you need, just run `make` to build Dynamo-RIO.

2.3 Running Dynamo-RIO

See README.

3 Current Project State

3.1 Visual Inspection of Code

In addition to the tree of INPROCESSS functions I created above, I have also performed visual checks of the code to try and see which source files are finished, in process, and yet to be started. This took into account any comments on the state of functions, whether code contained `ifdef ARM` which would indicate the code had at least been started to be adapted, if the code contained empty or dummy functions and the general look of the code. This resulted in the following:

core **Arm** Along with those specifically stated are several empty files.

- **arch.c** needs converting
- **arch.h** needs converting
- **arch_exports.h** Some dummy functions. Some x86.asm references?
- **arm.s** doesn't look finished
- **decode.c** contains both ARM and x86 in the same file. looks unfinished.
- **decode.h** as above
- **decode_fast.c** as above
- **decode_fast.h** as above
- **decode_table.c** done?
- **disassemble.c** not started
- **emit_utils.c** not finished. not started?
- **encode.c** not started
- **instr.c** done?
- **instr_create.h** not started
- **instrument.c** not finished?
- **instrument.h** not finished - Needs relevent DRAPI comments along with some code
- **interp.c** not finished
- **mangle.c** not finished
- **proc.c** not finished
- **proc.h** not started

lib **dr_config.h** may need some editing

- **genapi.pl** needs ARM integration
- **globals_shared.h** needs ARM integration
- **hotpatch_interface.h** needs ARM integration

linux **os.c** may need ARM integration

- **os_exports.h** may need ARM integration
- **signal.c** needs ARM integration
- **syscall.h** not finished?

On top of this, I also have the following list in my notebook under the title "changed files". A list of files that show signs of having been altered. I can't quite remember why I didn't give them the same treatment as the files above but the list will probably still be of some use.

CMakeLists.txt Note: I have modified this myself somewhat to allow for both native and cross compilation.

configure_defines.h

configure.h

configure_temp.h

core ldscript

CMakeLists.txt

globals.h

hotpatch.c

module_list.c

utils.c

vmareas.c

lib libdrpreload.so

make DynamoRIOConfig.cmake.in

3.2 Stephen Barton's Project Report

From reading Stephen Barton's project report Dynamo-RIO: Process Virtualisation for ARM, I managed to gather the following information about the current state of the project.

- An ARM instruction decoder has been written and incorporated, in part, into Dynamo-RIO.
- An ARM instruction encoder was also written but was only produced as a standalone program. A partialy completed version of this is available in the codebase that I have been told will handle "40 of the most common instructions". Apparently, adding new instructions is as simple as adding the relevant mask to the table.
- Not all ARM instructions have been supported. For example, those that update the CPSR with the format `<instr>S`.
- The re-implementation of the x86 functions has been partially done.
- It would also seem that an approach was taken to remove all the x86 functions and re-instate them as they were re-implemented for ARM to provide a side-by-side comparison.

3.3 Code Modified by James Sandford

With a fairly small number of dummy functions (`opnd_ceate_far_instr`, `opnd_create_instr`, `opnd_get_instr`, `opnd_get_segment_selector`, `opnd_is_far_instr` and `opnd_is_instr`) in texttttcore/Arm/instr.c, I was able to build the software and get some debugging output of the software traversing it's intialisation routines. With a little more effort, I discovered a bug in the interface between the C code and the

ARM assembly code and got more output. A copy of this output is available in the `logs/runLog1_9_11` file.

I originally interpreted some of this output as decoding due to it's apperance at a glance being similar to that of the output from the standalone decoder. This is not the case though. No decoding of instructions has been carried out by myself within the confines of Dynamo-RIO. I traced the reason for this to the clients I had assumed were compiled for ARM being compiled for x86. This meant I had to re-visit the build scripts for clients and, by extention, for the includes files. I added a little code to genapi.pl to add ARM support. It had previously just read from the x86 directory, not the ARM directory. Neither did it check for ARM includes and ifdefs. Although I don't believe I touched anything to do with it, a bug in this script that resulted in replication of code now seems to have dissapeared.

The next problem was one in wich CMake would notice a change in the compiler being used and would decide the best action would be to delete the CMake settings file and start with the standard ones... I eventually tracked this down to the building of clients being a seperate CMake project. The project() function would modify the relevent variables causing the problem. I came up with several possible solutions in accordance with the recomendations in the CMake documentation but none of these would have worked because of the ordering of option selection and the such. The solution I went with is the only one that I found to work. If we are not already within a CMake project, a new one will be created. Otherwise we just build everything as one project. This is not the right way of doing it so if anyone can find a way to keep the projects seperate, please implement it.

The current state of the project is that it will try to build the sample clients. The problem right now being that some of the source files for the "include" header files are not finished yet. Some functions are missing along with some of the DRAPI comments required for the genapi.pl script to copy things to the right place. There are also some errors popping up about miss-matched IFDEF statements. This may be a problem with the source files or the genapi.pl script. I am not certain which but I'm assuming the source files. I would recomend completing these as the next task. This would then allow you to build clients and thus test Dynamo-RIO properly.

Once Dynamo-RIO may be run nativly with an ARM client, I believe output may finally reach the decoder. It will then almost certainly fail as several of the functions in regards to the hashtable and Dynamo-RIO gaining full control of the code under test appear to be missing.

3.4 Scripts Created by James Sandford

3.4.1 DRTest

This is a small shellscript the runs make before copying the required to an ARM device (a machine called rightarm in this case) and then runs Dynamo-RIO. It also redirects the output to `logs/runLog` on the local machine. I'm using it with a trusted machine aliased in my ssh config but you can always modify the ssh lines to your machine and with the relevant username. Just set up your CMake config with `cmake -i` (if your config isn't changing, you only need to do this once) and then run `./DRTest` each time you want to test.

4 Other Notes

Other notes worth mentioning follow.

- `read_instruction` in core/Arm/decode.c is listed as done but seems to have a large amount of place-holders.
- The entry point at which Dynamo-RIO latches onto a process isn't immediately obvious. Initialization begins with the `_init` function in core/dynamo.c from where specific initialization routines are called. The `_init` function is called from the dynamic library which contains Dynamo-RIO just before the code under test is started. It is because of this that only dynamic code can be controlled by Dynamo-RIO.
- There are some basic programs for testing in the top level MiscCode folder. It is importand to note that you should NOT use `as` to assemble the ARM assembly code as this will result in non-dynamic code that Dynamo-RIO cannot latch on to. Use `arm-none-linux-gnueabi-gcc -nostartfiles <sourcefile>`.
- The jobs that need doing are not just those listed as INPROCESSS or ADDME. But those that are listed are a good starting place and will lead you to most, if not all, the other jobs needed to complete the project.