**Motivations**

1. Accelerating closed source programs by rewriting code with better scheduling.
2. Recovering the algorithm of a program without any scheduling code
3. Recompilation of filters into different targets.
4. Source code recovery of image processing filters.

For detailed list of all the algorithms and heuristics we use in this system read the design doc.

**File structure**

* preprocess - code\_diff, filter\_funcs
* postprocess – buildex, msummary
* dr\_clients – refer dr\_clients section details
* common – common source files
* tests – c tests, halide tests, asm tests
* logs – <safe to delete these files – only for debugging purposes>
* generated\_files - <do not delete these – can be used for testing>
* documentation
* run – contains the batchfiles for running, building the entire set of tools
* images

**Conventions for tools and clients**

**Standard files in a given tool folder**

* src
* include
* bin32 / bin64 - generated
* build32 / build64 – generated
* CMakelists.txt
* lib (any third party libs) – only if needed
* utility
  + clean.bat
  + build.bat <arch> <config>
  + run.bat <arch> <arguments> - arguments should be in quotes
  + run\_tests.bat <number> - this is for testing; pre-populate the arguments and then call the number of the test you need to run

**Command line processing conventions**

* -<name\_of\_command> value
* -help – a common command line switch that will output the usage
* -help <client\_names> - (DR only) will give out the arguments the clients are expecting

**Generated file naming conventions**

* <client\_name | executable\_name>\_<executable>\_<other\_info>\*.log

\*other info – thread id, image names etc.

**Common tasks and associated files**

* meminfo.cpp - Data structure population, memory region coalescing, defragmenting and getting memory regions out
* image\_manipulation.cpp – this file contains all the image manipulation and information retrieval common tasks
* commandline\_parse.cpp – this file does the command line argument parsing for the CPP files

These should be included into the project when building. Rather than a static library it is more convenient for now.

**Pre-process tools**

1. code\_cov.bat <arch>

This will dump the code coverage files into the output\_files folder.

File names – drcov.<process\_name>.<process\_id>.proc.log (files are created for each process – not thread)

1. code\_diff

First run drcov to get bbinfo for diffing; the idea is that running memtrace, profiling on a GUI application ifself would be too heavy; so first isolate the code and then profile it and identify the function entry points which you should keep track of to produce the instrace. So if you are going to use diff method for profiling do not profile with and without filter and give it to the filter\_funcs tool. It would be too heavy. Instead generate code coverage and then use code\_diff to generate a filter file which you can use for profiling.

* Inputs – the two code coverage files – obtained by the drcov client
* Outputs – filter file that can be used for profiling

Command line arguments (folders are all standardized)

* First – first file
* Second – second file
* Exec – this is the executable that was run to get the code coverage files

1. filter\_funcs

This will generate filter files for instrace, given basic block profile, images which they belong to and memtrace information.

* Modes of operation

1. diff basic block profile + 1 image
2. basic block profile full (2) + 2 images
3. basic block profile full + 1 image (this just gives out an approximate guess)

* Inputs – basic block profile, memtrace and images
* Outputs – filter\_file (function based filtering) + likely app\_pc may be writing to the images.

Command line arguments (folders are all standardized)

* mode – diff, oneim, twoim
* images – give out the image names in sequence ( we know the folder images reside in – so no problem)
* exec – this is the executable which was run to generate the profile

**DR\_client arguments and conventions**

DR clients are written as pluggable separate files carrying different instrumentations which can be invoked as a chain. It uses ‘drmgr’ multi-instrumentation manager to manage all the instrumentation passes that take place. The user can invoke any instrumentation pass he/she wants by specifying the name of the pass to DR client’s command line argument list.

-logdir is a special command line argument passed to the client, which if debug is passed will print out what clients are doing to it.

-<pass name> <arg1> <arg2> …. -<pass name> <arg1> <arg2>

**dr\_clients**

All command line arguments to the clients should be entered in order. Output files will follow the file naming conventions.

1. cpuid – modifies the return value of the CPUID assembly instruction.

ARGS – none

1. functrace – keeps track of the function information (entry points) – any other pass can query function entry points.

ARGS – <filter file> <filter mode>

1. Funcwrap – this wraps around a function, providing facilities for function based filtering

ARGS – <filter file> (no need for filter mode – this does function based filtering)

1. Inscount – counts the total dynamic instructions executed with in the program

ARGS – <filter file> <filter mode>

1. **Instrace** – this gives out a complete dynamic instruction trace of the executed program

ARGS - <filter file> <filter mode> <output\_folder> <static\_info\_size> <instrace mode>

Instrace modes –

* 1. Operand trace (1) – this traces only the instructions + operands that are within the filtered instructions. This does not instrument the code at all. This is just an analysis routine.
  2. Opcode trace (2) – this traces all the opcodes used with in the filtered region. This is again just analysis, no instrumentation is done.
  3. Disassembly trace (3) – prints out all the dynamic instructions as assembly mnemonics.
  4. Instruction trace (4) – the real instruction trace that is parseable by buildex tool

1. **Memtrace** – this gives out a memory trace of the executed program

ARGS - <filter file> <filter mode> <output folder>

1. **Profile\_global** – this gives out a profile for the executed program including bb frequency, callees, callers and dynamic control flow related information

ARGS – <filter file> <filter mode> <output folder> <threshold>

1. ***Memdump*** – this dumps the memory which can be read by a tool for processing.

ARGS – <filter file> <filter mode> <output folder>

\*Boldfaced passes outputs files – should be given the folder to output, filenames will be based on the convention mentioned earlier. \*Italics are yet to be implemented.

All clients support filtering based instrumentation based on following criteria. Filtering is handled by utilities.cpp

1. Filter based on the module names
2. Filter based on specified bbs in the given modules
3. Filter based on a range of addresses in the given modules
4. Filter based on functions
5. Filter based on the not being in the listed modules (opposite to (1))
6. ***Filter based on the app\_pc in the given modules***
7. No filtering

Filter file specification – this is the format of the filter file that should be given to the DR client

number of modules

<module name>

number of addresses

addresses

<module name>

number of addresses

addresses

...

**Post-process tools**

1. buildex

This will construct the memory layout based on the instrace, then select a suitable memory location, build a dependency tree for it and abstract it out suitable for printing out a halide file.

* Inputs – instrace, input and output images, app\_pc to select a suitable mem locations
* Output – dot files for the concrete tree, compound concrete tree, abstract tree and the halide output.

Command line arguments (folders are all standardized)

* Exec - this is the executable which was run to generate the instrace
* Start – line number to start
* End – line number to end trace
* Dest – if you are giving the start

**Logical system diagram**

Refer the power point slides

**Generated files**

* Filter\_files
* halide\_files
* output\_files

**Tests**

* c\_tests
  + basic tests – arithmetic, calls, convoluted, data\_parallel, ifelse, loop\_red, producer\_consumer, recursive
  + image processing filters – blur, invert
* halide\_tests
  + halide\_blur
  + halide\_rotate
  + halide\_input\_dep\_filter (bilateral filter without spatial component)
  + halide\_resize
  + halide\_snake
  + halide\_composition of filters
  + halide\_tile\_copy
  + all the above filters with various scheduling
* asm\_tests
  + Yet to decide
* Photoshop examples