MIR User Guide

November 23, 2016

1 Introduction

MIR is an experimental task-based runtime system library written using C99. Core features of MIR include:

- Support for a capable subset of OpenMP 3.0 tasks and parallel for-loops.
- Competitive performance for medium-grained task-based programs.
- Flexible, high performance task scheduling and data distribution policies. Examples include locality-aware scheduling and data distribution for NUMA systems and work-stealing scheduling for multicore systems.
- Detailed per-task performance profiling and support for Grain Graph [1] visualization.

2 Intended Audience

MIR is intended to be used by advanced task-based programmers. Knowledge of OpenMP compilation and role of runtime system is required to use and appreciate MIR. Since MIR is experimental, some user interfaces may be unpolished. Be prepared to get your hands dirty.

3 Requirements

MIR is built and tested on modern (year 2012 and later) Linux-based systems.

In order to build and use MIR for task-based program execution, you will minimally require:

- A machine with x86 (bit size irrelavant) architecture
- Linux kernel later than January 2012
- GCC
- Python
- GNU Binutils
- Scons build system
- Check, a unit testing framework
- R
- These R packages:
  - data.table

Enabling core features such as OpenMP support, per-task profiling, and NUMA-specialized execution requires:

- Libraries libnuma and numactl
- GCC with OpenMP support
- PAPI
- Paraver from BSC
- Intel Pin sources
- These R packages:
  - optparse
  - igraph
  - RColorBrewer
  - ggplot2 and reshape2
  - gdata, plyr, dplyr, and scales
  - pastecs

4 Source Structure

The MIR source repository is easy to navigate. Files and directories have familiar, purpose-oriented names. The directory structure is:

```
. : MIR_ROOT
|__docs : documentation
|__src : runtime system sources
   |__scheduling : scheduling policies
   |__arch : architecture specific code
|__scripts
   |__profiling : all things related to profiling
      |__task
         |__for_loop
      |__thread
|__tests : test suite
|__examples : example programs
```

5 Licensing

MIR is released under the Apache 2.0 license. As long as the MIR native library interface is used to compose task-based programs, the Apache 2.0 license is binding.

However, OpenMP support is enabled through a GPL (v3.0) implementation of the GNU *libgomp* interface. Therefore a combination of Apache 2.0 License and GPL is applicable when OpenMP programs are linked with MIR. Understanding the implications of the combination is the responsibility of the user.

6 Build

Follow below steps to build the basic runtime system library.

- Set MIR_ROOT environment variable.

```
$ export MIR_ROOT=<MIR source repository path>
```

> Tip: Add the export statement to .bashrc to avoid repeated initialization.

- Build.

```
$ cd $MIR_ROOT/src
$ scons
```

6.1 Enabling NUMA systems support

To enable data distribution and locality-aware scheduling on NUMA systems, follow below instructions.

- Install the libraries libnuma and numactl.
- Create an empty file called HAVE_LIBNUMA.

```
$ touch $MIR_ROOT/src/HAVE_LIBNUMA
```

- Clean and rebuild MIR.

```
$ cd $MIR_ROOT/src
$ scons -c && scons
```

6.2 Enabling OpenMP support

To enable support for OpenMP, follow below instructions.

- Download the GPL implementation of the libgomp interface from the GitHub repository https://github.com/anamud/mir-omp-int. Point the environment variable `MIR_OMP_INT_ROOT` to the download location.
- Link the GPL implementation to the source directory and rebuild MIR.

```
$ cd $MIR_ROOT/src
$ ln -s $MIR_OMP_INT_ROOT/mir_omp_int.c mir_omp_int.c
```

- Clean and rebuild MIR.

```
$ cd $MIR_ROOT/src
$ scons -c && scons
```

7 Testing

Run tests in MIR_ROOT/tests. Make it a habit to run tests for each change to source repository. Add new tests if necessary.

```
$ cd $MIR_ROOT/tests
$ ./test-all.sh | tee test-all-result.txt
```

8 Example Programs

Run example programs in MIR_ROOT/examples.

```
$ cd $MIR_ROOT/examples/OMP/fib
$ scons -u
$ ./fib-opt.out
```

> Tip: A dedicated suite of benchmark programs for testing MIR is available upon request.

9 OpenMP Programming

OpenMP support is restricted to the following interfaces from OpenMP version 3.0:

- Task creation: `task shared(list) private(list) firstprivate(list) default(shared|none)`
- Task synchronization: `taskwait`
- Parallel block: `parallel shared(list) private(list) firstprivate(list) num_threads(integer_expression) default(shared|none)`
- Single block: `single`
- For-loop: `for shared(list) private(list) firstprivate(list) lastprivate(list) reduction(reduction-identifier:list) schedule(static|dynamic|runtime|guided[,chunk_size])`.
- Combined parallel block and for-loop: `parallel for`
- Serialization: `atomic`, {`critical [,name]`}, `barrier`
- Runtime functions: `omp_get_num_threads, omp_get_thread_num, omp_get_max_threads, omp_get_wtime`
- Environment variables: `OMP_NUM_THREADS, OMP_SCHEDULE`

9.1 Tips for writing MIR-supported OpenMP programs

- Use `taskwait` explicitly to synchronize tasks. Do not expect implicit task synchronization points within thread barriers.

- Avoid distributing work to threads manually. Let the runtime system schedule tasks on threads.
- Study example and test programs.
- You can expect a compiler/runtime error when a non-supported interface is used.

9.2 GCC restriction

OpenMP support is restricted to programs compiled using GCC. MIR intercepts GCC translated calls to GNU libgomp when linked with OpenMP programs.

10 Native Interface Programming

MIR interfaces can be directly used to compose task-based programs. Look at the header file mir_public_int.h in MIR_ROOT/src for interface details and programs in MIR_ROOT/examples/native for usage examples.

11 Compiling and Linking

A quick way to compile and link with programs is to reuse the SConstruct or SConscript files of example programs in MIR_ROOT/examples/.

If compiling manually, add `-lmir-opt` to `LDFLAGS`. When profiling instructions of programs, enable MIR to profile outline function calls correctly by adding `-fno-inline-functions -fno-inline-functions-called-once -fno-optimize-sibling-calls -fno-omit-frame-pointer -g` to `CFLAGS` and `CXXFLAGS`.

12 Runtime Configuration

MIR has several runtime configurable options that can be set using the environment variable `MIR_CONF`. Set the `-h` flag to see available configuration options.

```
$ MIR_CONF="-h" <invoke MIR-linked program>
...
-h (--help) print this help message
-w <int> (--workers) number of workers (including master thread)
```

```
-s <str> (--schedule) task scheduling policy. Choose among policies central, central-
    stack, ws, ws-de and numa.
-m <str> (--memory-policy) memory allocation policy. Choose among coarse, fine
    and system.
--inlining-limit=<int> task inlining limit based on number of tasks per worker.
--stack-size=<int> worker stack size in MB
--queue-size=<int> task queue capacity
--numa-footprint=<int> data footprint size threshold in bytes for numa scheduling
    policy. Tasks with data footprints below threshold are dealt to worker's private
    queue.
--worker-stats collect worker statistics
--task-stats collect task statistics
-r (--recorder) enable worker recorder
-p (--profiler) enable communication with Outline Function Profiler. Note: This
    option is supported only for single-worker execution!
...
```

Say you want to enable the coarse memory allocation policy and use 4 workers, then the configuration should be written as,

```
$ MIR_CONF="-w 4 --memory-policy=coarse" <invoke MIR-linked program>
```

12.1 Binding workers to cores

Threads created by MIR are called *workers*. The master thread is also a worker.

MIR creates and binds one worker per core by default. Hardware threads are always disregarded while binding. Binding is based on worker identifiers — worker thread 0 is bound to core 0, worker thread 1 to core 1 and so on.

The binding scheme can be changed to a specific mapping using the environment variable `MIR_WORKER_CORE_MAP`. Ensure `MIR_WORKER_EXPLICIT_BIND` is defined in `mir_defines.h` to enable explicit binding support. An example is shown below.

```
$ cd $MIR_ROOT/src
$ grep "EXPLICIT_BIND" mir_defines.h
#define MIR_WORKER_EXPLICIT_BIND
$ cat /proc/cpuinfo | grep -c Core
4
$ export MIR_WORKER_CORE_MAP="0,2,3,1"
$ <invoke MIR-linked program>
MIR_DBG: Starting initialization ...
MIR_DBG: Architecture set to firenze
```

```
MIR_DBG: Memory allocation policy set to system
MIR_DBG: Task scheduling policy set to central-stack
MIR_DBG: Reading worker to core map ...
MIR_DBG: Binding worker 0 to core 3
MIR_DBG: Binding worker 3 to core 0
MIR_DBG: Binding worker 2 to core 2
MIR_DBG: Worker 2 is initialized
MIR_DBG: Worker 3 is initialized
MIR_DBG: Binding worker 1 to core 1
...
```

13 Thread-based Profiling

MIR supports extensive and detailed thread-based profiling. Profiling data is obtained and processed using special scripts and stored mainly as CSV files. Columns names in CSV files are self-explanatory. Contact MIR contributors for clarifications about column names.

Thread states and events are the main performance indicators in thread-based profiling. Set the `--worker-stats` flag to get basic thread statistics in a CSV file called mir-worker-stats.

```
$ MIR_CONF="--worker-stats" <invoke MIR-linked program>
$ cat mir-worker-stats
```

MIR contains a tracing module called the *recorder* that produces time-stamped execution traces. Set the `-r` flag to enable the recorder and get detailed state and event traces in a set of mir-recorder-trace-*.rec files. Each file represents a worker. Inspect the files individually, or combine them for visualization on Paraver using a special script.

```
$ MIR_CONF="-r" <invoke MIR-linked program>
$ $MIR_ROOT/scripts/profiling/thread/rec2paraver.py \
mir-recorder-trace-config.rec
$ wxparaver mir-recorder-trace.prv
```

> Tip: Paraver configuration files for studying memory hierarchy utilization problems are placed in $MIR_ROOT/scripts/profiling/thread/paraver-configs.

To understand time spent by workers in individual states without using Paraver, use a special script to process mir-recorder-state-time-*.rec files created by the recorder.

```
$MIR_ROOT/scripts/profiling/thread/get-states.sh -w mir-worker-stats \
mir-recorder-state-time-*
$ cat state-summary.csv
$ head states.csv
```

13.1 Enabling hardware performance counters

MIR can read hardware performance counters through PAPI during task execution events. Events currently supported are task start and task end events. In particular, the task switch event is not supported. This means that counter readings will include effects of runtime system activity, system calls, interrupts etc that happened during task execution.

- Install PAPI.
- Set the PAPI_ROOT environment variable

  ```
  $ export PAPI_ROOT=<PAPI install path>
  ```

- Create a file called HAVE_PAPI in MIR_ROOT/src.

  ```
  $ touch $MIR_ROOT/src/HAVE_PAPI
  ```

- Enable additional PAPI hardware performance counters by editing MIR_ROOT/src/mir_recorder.c. In the example below, counters `PAPI_TOT_INS` and `PAPI_TOT_CYC` are enabled. Ignore the 0x0 value.

  ```
  $ grep -i "{PAPI_" $MIR_ROOT/src/mir_recorder.c
  {"PAPI_TOT_INS", 0x0},
  {"PAPI_TOT_CYC", 0x0},
  /*{"PAPI_L2_DCM", 0x0},*/
  /*{"PAPI_RES_STL", 0x0},*/
  /*{"PAPI_L1_DCA", 0x0},*/
  /*{"PAPI_L1_DCH", 0x0},*/
  ```

- Rebuild MIR.

  ```
  $ scons -c && scons
  ```

Performance counter readings will appear in the mir-recorder-trace-*.rec files created by the recorder during profiling. These files can either be viewed on Paraver or processed using a special script for manual analysis.

```
$ $MIR_ROOT/scripts/profiling/thread/get-events.sh mir-recorder-trace.prv
$ cat events-*-summary.csv
```

14 Task-based Profiling

MIR supports extensive and detailed task-based profiling. Profiling data is obtained and processed using special scripts and stored mainly as CSV files. Columns names in CSV files are self-explanatory. Contact MIR contributors for clarifications about column names.

Per-task metrics are first-class performance indicators in task-based profiling. Per-task refers to individual task instances whose count is typically much larger than the number of task definition sites in source code. For example, the Fibonacci number program (`MIR_ROOT/examples/OMP/fib`) defines two tasks in source code that together create 8193 task instances for the inputs n=45 and *depth cutoff*=12.

Set the `--task-stats` flag to obtain per-task statistics in a CSV file called mir-task-stats. The file contains raw data that should be processed using a special script before starting any analysis. The script straightens out the raw data and optionally derives metrics such as run-independent unique identifiers. Check out options by setting the `-h` flag.

```
$ MIR_CONF="--task-stats" <invoke MIR-linked program>
$ Rscript ${MIR_ROOT}/scripts/profiling/task/process-task-stats.R -h
$ Rscript ${MIR_ROOT}/scripts/profiling/task/process-task-stats.R \
-d mir-task-stats
$ head task-stats.processed
```

The processed file task-stats.processed is too large for manual inspection in a text editor. Crunch it with powerful data analysis tools such as R to derive useful information. The output of task statistics processing scripts process-task-stats.R and `merge-task-stats.R` can be summarized by a special script called summarize-task-stats.R.

```
$ Rscript ${MIR_ROOT}/scripts/profiling/task/summarize-task-stats.R \
-d task-stats.processed
$ cat task-stats.summarized
```

14.1 Advanced task-based metrics

Hardware performance counter readings collected by the recorder (Section 13) can be merged with processed task statistics using special scripts.

```
$ $MIR_ROOT/scripts/profiling/thread/get-events-per-task.sh \
mir-recorder-trace-*.rec
$ Rscript $MIR_ROOT/scripts/profiling/task/merge-task-stats.R \
```

```
-l task-stats.processed -r events-per-task-summary.csv -k task -c left
$ head task-stats.merged
```

Work deviation is a derived performance metric that requires comparing execution times of tasks under multithreaded execution. See the paper [1] for more details. Below is an example of how to calculate work deviation across 1 and 4 workers for the Fibonacci example program for inputs n=45 and*depth cutoff*=12.

```
$ MIR_CONF="--task-stats -w 1" ./fib-opt 45 12
$ Rscript $MIR_ROOT/scripts/profiling/task/process-task-stats.R \
-d mir-task-stats --lineage
$ mv task-stats.processed task-stats-w1.processed
$ MIR_CONF="--task-stats -w 4" ./fib-opt 45 12
$ Rscript $MIR_ROOT/scripts/profiling/task/process-task-stats.R \
-d mir-task-stats --lineage
$ Rscript $MIR_ROOT/scripts/profiling/task/compare-task-stats.R \
-l task-stats.processed -r task-stats-w1.processed \
-k lineage
$ Rscript $MIR_ROOT/scripts/profiling/task/merge-task-stats.R \
-l task-stats.processed -r task-stats.compared \
-k lineage
$ head task-stats.merged
```

14.1.1 Instruction-level task profiling

MIR provides a Pin-based instruction profiler for tasks called the *Outline Function Profiler* (OFP). The OFP traces instructions executed within outline functions of tasks. Outline functions are inserted by the compiler as wrappers for task structure blocks. Read paper [2] for more details.

The limitations of OFP are:

- Instructions of system calls called within the outline function are not traced due to technology limitations.
- Supports OpenMP 3.0 task-based programs only. Programs with non-task features such as parallel for-loops, manual division of work among parallel blocks, sections are not supported.

Follow below steps to build the OFP.

- Download Intel *Pin* sources and set associated environment variables.

  ```
  $ export PIN_ROOT=<Pin source path>
  $ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\
  :$PIN_ROOT:$PIN_ROOT/intel64/runtime
  ```

- Edit `PIN_ROOT/source/tools/Config/makefile.unix.config` and add `-fopenmp` to variables `TOOL_LDFLAGS_NOOPT` and `TOOL_CXXFLAGS_NOOPT`
- Build.

```
$ cd $MIR_ROOT/scripts/profiling/task
$ make PIN_ROOT=$PIN_ROOT
```

The OFP is technically a Pin Tool. View its options using the `-h` flag.

```
$ $PIN_ROOT/intel64/bin/pinbin -t $MIR_ROOT/scripts/profiling/task/obj-intel64/
    mir_of_profiler.so -h -- /usr/bin/echo
...
-of outline functions (csv)
-cf functions called from outline functions (csv)
-df dynamically library functions called from outline functions (csv)
-pr output prefix [default mir-ofp]
...
```

Runtime system function calls made within tasks are not profiled and attributed to tasks by default. If profiling and attribution of runtime system function calls to tasks is required, provide `-ni` flag argument.

OFP works in tandem with the runtime system. To couple OFP and the runtime system together, enable the `-p` flag in MIR_CONF to enable handshaking.

The profiler requires single-threaded execution of the profiled program. Provide `-w 1` in MIR_CONF while profiling.

Information from the profiler becomes more meaningful when correlated with task statistics information. Provide `--task-stats` in MIR_CONF while profiling.

Use script `of_finder.py` to find outline functions and functions called within outline functions.

Create a handy shell function for invoking the profiler and to enable task statistics collection.

```
$ type mir-inst-prof
mir-inst-prof is a function
mir-inst-prof ()
{
    MIR_CONF='-w 1 -p --task-stats --single-parallel-block' ${PIN_ROOT}/intel64/
      bin/pinbin -t ${MIR_ROOT}/scripts/profiling/task/obj-intel64/mir_of_profiler.so
       "$@"
}
```

The profiler takes approximately 36X the time to execute the program on a single core and produces three CSV files – `mir-ofp-instructions`, `mir-ofp-events` and `mir-task-stats`.

> IMPORTANT: The Task Performance Extractor should be executed natively on the host machine. It does not work reliably when executed on virtual machines (for example, on the PaPP Development Environment VM) due to technology limitations.

14.1.2 Composition

The Task Performance Extractor is designed for OpenMP 3.0 task-based programs only. *OpenMP programs with non-task features such as parallel for-loops, division of work among parallel blocks and sections are not supported.*

Follow guidelines below to compose task-based OpenMP programs that can be input to the Task Performance Extractor.

- Think solely in terms of OpenMP 3.0 tasks.
  - Use the `task` construct to parallelize work.
  - Use clauses `shared`, `firstprivate` and `private` to indicate the task data environment.
  - Use the `taskwait` construct to synchronize tasks.
  - Use `taskwait` explicitly. Do not expect implicit taskwaits to be added by the compiler or runtime system.
- Fully avoid thinking in terms of threads.
  - Use the `parallel` construct only to create a team of threads.
  - Do not specify the number of threads using either the `num_threads` clause or the `OMP_NUM_THREADS` environment variable. Let the runtime system decide on the number of threads required for execution.
  - Do not use the `parallel` construct to share work.
  - Do not use the `barrier` construct to synchronize threads. Do not expect implicit barriers at the end parallel blocks.
- Fully avoid all non-task constructs such as `for` (parallel for-loops), `parallel` (parallel section) and `section` (another form of work-sharing).

- Use the `critical` or `atomic` construct for serialization within tasks.
- Use GCC atomic builtins for flushing within tasks.
- Avoid new task features from OpenMP 4.0 such as data-dependent implicit synchronization of tasks.

The code template below can be a used as a boiler-plate for producing task-based OpenMP programs supported by the Task Performance Extractor.

```
int main(int argc, char *argv[])
{

#pragma omp parallel
{
   #pragma omp single
   {
      // Create a master task.
      // A master task enables profiling of parallelization code.
      #pragma omp task
      {
         // Parallelize work here using tasks.
         // Make sure to synchronize with the tasks using taskwait.
      }
      // Wait for master task to finish
      #pragma omp taskwait
   } // omp single end
} // omp parallel end

   return 0;
}
```

14.1.3 Third-party function calls

The Task Performance Extractor can only profile calls made within tasks to user-level functions. System calls cannot be profiled and attributed to tasks.

Runtime system function calls made within tasks are not profiled and attributed to tasks by default.

14.1.4 Compilation

Program compilation should ensure outline functions of tasks can be obtained by inspecting object files. Provide the following flags to GCC during

program compilation: `-O1 -fno-inline-functions -fno-inline-functions-called-once -fno-optimize-sibling-calls -fno-omit-frame-pointer`

Additionally, ensure that object files produced during compilation are available for inspection. This can be done by an explicit two-step compilation process. First, compile source files into objects. Next, link the object files together to create the executable.

We show an example of the compilation process for the PaPP use case *UC_RadarDet* below.

```
$ cd $PAPP_SVN/Code/WP2/UC_RadarDet/
$ g++ -Wall -fopenmp -DFIXED_POINT=16 -O1 -fno-inline-functions -fno-inline-
    functions-called-once -fno-optimize-sibling-calls -fno-omit-frame-pointer -
    I$MIR_ROOT/src -c radar_sigproc.c kiss_fft/kiss_fft.c kiss_fft/kiss_fft.h kiss_fft/
    _kiss_fft_guts.h
$ g++ -o radar_sigproc radar_sigproc.o kiss_fft/kiss_fft.o -lm -lpthread -lpapi -
    L$MIR_ROOT/src -lmir-opt
```

14.2 Usage

Using the Task Performance Extractor is a simple process consisting of the following three steps:

1. Identify which tasks to profile in the compiled program.
2. Profile architecture independent metrics of identified tasks by executing the program.
3. Merge architecture independent metrics obtained during profiling to produce the task graph structure in a post-processing step.

We now explain each of the steps and demonstrate for the PaPP use case UC_RadarDet.

1. To identify which tasks to profile in the compiled program, use the script `MIR_ROOT/scripts/profiling/task/of_finder.py`. The script takes object files as input and outputs three lists:
   - A list called CHECKME_OUTLINE_FUNCTIONS containing names of task outline functions defined in the object files.
   - A list called CHECKME_CALLED_FUNCTIONS containing names of functions potentially called from inside the outline functions.

- A list called CHECKME_DYNAMICALLY_CALLED_FUNCTIONS containing names of functions potentially called dynamically from inside the outline functions.

Inspect the three lists with your local OpenMP expert and ensure there are no ambiguities. Examples of ambiguities include non-outline funcitons in `CHECKME_OUTLINE_FUNCTIONS`, duplicated/common items in lists or empty lists. In the typical case, the lists are proper by default and inspecting them is just a quick sanity check. After confirming that the lists are free of ambiguities, export them into the shell. This can be done using backticks on the output produced by the `-e` option of the `of_finder.py` script.

Identifying tasks to profile in UC_RadarDet is shown below.

```
$ $MIR_ROOT/scripts/profiling/task/of_finder.py ./radar_sigproc
CHECKME_OUTLINE_FUNCTIONS=
    _ZL7kf_workP12kiss_fft_cpxPKS_miPiP14kiss_fft_state._omp_fn.0,...
CHECKME_CALLED_FUNCTIONS=
    _ZL8kf_bfly2P12kiss_fft_cpxmP14kiss_fft_statei,...
CHECKME_DYNAMICALLY_CALLED_FUNCTIONS=memcpy,pthread_attr_init
    ...
$ `$MIR_ROOT/scripts/profiling/task/of_finder.py -e ./radar_sigproc`
```

2. The next step is to profile the program and extract architecture independent metrics of instances of tasks identified previously. Profiling is performed by the custom Pin tool and the MIR runtime system in tandem. We first define a convenient shell function called `mir-inst-prof` that encapsulates profiling arguments to the custom Pin tool and the MIR runtime system. Details of `mir-inst-prof` are shown below.

```
$ type mir-inst-prof
mir-inst-prof is a function
mir-inst-prof ()
{
   MIR_CONF='-w 1 -p --task-stats --single-parallel-block' ${PIN_ROOT}/
     intel64/bin/pinbin -t ${MIR_ROOT}/scripts/profiling/task/obj-intel64/
     mir_of_profiler.so "$@"
}
```

We invoke the function `mir-inst-prof` with the program and lists exported by `of_finder.py` as inputs. The function returns in approximately 36X the time to execute the program on a single core and produces three files – `mir-ofp-instructions, mir-ofp-events` and `mir-task-stats`. Here is an example for UC_RadarDet:

```
$ mir-inst-prof -of $CHECKME_OUTLINE_FUNCTIONS -cf
    $CHECKME_CALLED_FUNCTIONS -df
    $CHECKME_DYNAMICALLY_CALLED_FUNCTIONS -- ./radar_sigproc
    input.csv output.csv
$ ls -l mir-task-stats mir-ofp-events mir-ofp-instructions
-rw-rw-r-- 1 ananya ananya 282K Jun 16 13:17 mir-ofp-events
-rw-rw-r-- 1 ananya ananya 3.3M Jun 16 13:17 mir-ofp-instructions
-rw-r--r-- 1 ananya ananya 518K Jun 16 13:17 mir-task-stats
```

If profiling and attribution of runtime system function calls to tasks is required, provide `-ni` flag argument to `mir-inst-prof`.

3. The last step is to merge the output of the profiler to produce a task graph structuture. The step simply involves executing a series of merge scripts. An example for UC_RadarDet follows.

```
$ Rscript $MIR_ROOT/scripts/profiling/task/process-task-stats.R -d mir-task-
    stats
...
Wrote file: task-stats.processed
$ Rscript $MIR_ROOT/scripts/profiling/task/merge-task-stats.R -l task-stats.
    processed -r mir-ofp-instructions -k task - o merged-task-perf
...
Wrote file: merged-task-perf
```

The task graph structure is tabulated in the file `merged-task-perf`. This file and other intermediate profiling output files are passed on to the Task Execution Simulation component (see Section **??**) for further analysis leading to performance prediction.

The three usage steps can be integrated into a single script. Below is an example for UC_RadarDet.

```
$ $($MIR_ROOT/scripts/profiling/task/of_finder.py -e $(find . -name "*.o" -print0 |
    xargs --null))
$ mir-inst-prof -of $CHECKME_OUTLINE_FUNCTIONS -cf
    $CHECKME_CALLED_FUNCTIONS -df
    $CHECKME_DYNAMICALLY_CALLED_FUNCTIONS -- ./radar_sigproc input.
    csv output.csv
$ Rscript $MIR_ROOT/scripts/profiling/task/process-task-stats.R -d mir-task-stats
$ Rscript $MIR_ROOT/scripts/profiling/task/merge-task-stats.R -l task-stats.
    processed -r mir-ofp-instructions -k task - o merged-task-perf
```

14.2.1 Visualization

14.3 Grain Graph Visualization

Processed task statistics can also be visualized graphically using a recent visualization method called the *Grain Graph* [1]. See the GitHub repository https://github.com/anamud/grain-graph to understand how.

MIR has a nice graph plotter which can transform task-based profiling data into task graphs. The generated graph can be visualized on tools such as yEd and Cytoscape. To plot the fork-join task graph using task statistics from the runtime system:

```
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d task-stats.
    processed -p color
```

> Tip: The graph plotter will plot in gray scale if `gray` is supplied instead of `color` as the palette (`-p`) argument. Critical path enumeration usually takes time. To speed up, skip critical path enumeration and calculate only its length using option `--cplengthonly`.

The graph plotter can annotate task graph elements with performance information. Merge the instruction-level information produced by the instruction profiler with the task statistics produced by the runtime system, for the same run, into a single CSV file. Plot task graph using combined performance information.

```
$ Rscript ${MIR_ROOT}/scripts/profiling/task/process-task-stats.R -d mir-task-stats
$ Rscript ${MIR_ROOT}/scripts/profiling/task/merge-task-performance.R -l task-
    stats.processed -r mir-ofp-instructions -k "task" -o mir-task-perf
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d mir-task-perf -p
     color
```

TODO: Instructions to use the full task graph profiler.

> Tip: Mapping profiling data to visual attributes of the task graph is essential to understand the program structure and problems quickly. If you are using YEd to see the task graph, then import and apply property mapper settings from files called `*-property-map-yed.cnfx` under `$MIR_ROOT/scripts/profiling/task/`, and then layout the graph using the hierarchical layout.

Processed information can be manually inspected or visualized on a fork-join task graph.

```
$ Rscript ${MIR_ROOT}/scripts/profiling/task/process-task-stats.R -d mir-task-stats
      --lineage
$ cat task-stats.info
$ head task-stats.processed
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d task-stats.
     processed
```

TODO: Explain file contents.

14.4 Profiling Case Study

This section is yet to be written. See paper *Grain Graphs: OpenMP Performance Analysis Made Easy* published in PPOPP16 for profiling and visualization case studies that use MIR.

References

[1] **Muddukrishna, Ananya**, P. A. Jonsson, A. Podobas, and M. Brorsson, "Grain graphs: OpenMP performance analysis made easy," 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'16), 2016.

[2] **Muddukrishna, Ananya**, P. A. Jonsson, and M. Brorsson, "Characterizing task-based OpenMP programs," *PLoS ONE*, vol. 10, no. 4, 04 2015.