MIR User Guide

Ananya Muddukrishna, Peter A. Jonsson, and other contributors
to the anamud/mir-dev GitHub repository

March 3, 2017

Contents

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 for advanced task-based programming experimentation. Knowledge of OpenMP compilation and role of runtime system is required to use and appreciate MIR. Some experimental user interfaces may not be as refined as others. 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 irrelevant) 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 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 Citation

If you use MIR in your work, please cite one (or all) of these related papers that shaped MIR:

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

- A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, “Locality-aware task scheduling and data distribution on NUMA systems,” in *OpenMP in the Era of Low Power Devices and Accelerators*. Springer Berlin Heidelberg, 2013, pp. 156–170.

7 Build

Fire up a BASH terminal and follow below steps to build the basic runtime system library.

- Download MIR from the GitHub repository https://github.com/anamud/mir-dev. Point environment variable MIR_ROOT to the download directory.

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

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

- Build.

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

7.1 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
```

7.2 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 directory.
- 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
```

8 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
```

9 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.

10 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`

10.1 Tips for writing supported OpenMP programs

Follow the tips below to write OpenMP 3.0 programs supported by MIR.

- 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.
- Don't mix tasks and for-loops. Write exclusively task-based or for-loop-based programs.
- Study example and test programs.
- You can expect a compiler/runtime error when a non-supported interface is used.

10.2 GCC restriction

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

11 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.

12 Compiling and Linking Programs

A quick way to compile and link with programs is to reuse the SConstruct or SConscript files of example programs in MIR_ROOT/examples/. The scripts have configurations to produce release, debug and profiling friendly executables.

If compiling manually, add `-lmir-opt` to `LDFLAGS`.

13 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 enable worker statistics
```

```
--task-stats enable 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>
```

13.1 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
...
```

13.2 Scheduling policies

MIR scheduling is configurable through scheduling policies. The scheduling policy is set using the `--schedule` or `-s` option in `MIR_CONF` and cannot be changed during execution.

The scheduling policies available are:

central : This policy adds all created tasks to a single queue. Idle workers remove the oldest tasks from the queue. Tasks are immediately executed if the queue is full. The capacity of the queue is set in mir_defines.h. Workers contend with each other to add and remove tasks from the single queue. This creates a performance bottleneck when the number of workers are large.

central-stack : This is similar to the central policy except that the container to hold tasks is a stack. Newest tasks are removed first by workers.

ws : This is a work-stealing policy based on double-ended queues. Each worker adds and removes tasks from a local queue. When tasks cannot be found in the local queue, workers steal tasks from local queues of other workers. This policy is scalable to a large number of threads when steals are rare. Tasks are immediately executed if the queue is full. The capacity of the queue is set in mir_defines.h

ws-de : This is similar to the ws policy except that the container to hold tasks is a lock-free double-ended queue designed by Chase and Lev [4].

numa : This is a locality-aware policy for NUMA systems that works well with prudent data distribution. The policy uses a double-ended queue per NUMA node. Each task is added to the queue that has the least latency to data not in the L3 cache. Idle workers remove tasks from the queue local to their own NUMA node first. If the queue is empty, then workers steal from queues local to other NUMA nodes starting from the closest node. Tasks are immediately executed if the queue is full. The capacity of the queue is set in mir_defines.h. See the paper [5] for details.

Note: MIR does not support *untied* tasks model of OpenMP.

13.3 Memory allocation policies

MIR enables data distribution through memory allocation policies. Data distribution refers to distributing memory pages to NUMA nodes. The memory allocation policy, set using the `--memory-policy` or `-m` option in `MIR_CONF`, controls how the pages are distributed.

The memory allocation policy works only on memory allocation operations made using the functions `mir_mem_pol_reset`, `mir_mem_pol_allocate`, and `mir_mem_pol_release`. See mir_public_int.h for exact function signatures.

Available memory allocation policies are:

system : This delegates memory page distribution to the OS.

fine : This distributes memory pages to all NUMA nodes page-wise round-robin. If `MIR_MEM_POL_RESTRICT` is defined in mir_defines.h, then pages are distributed to NUMA nodes local to workers.

coarse : This distributes all memory pages requested by a `mir_mem_pol_allocate` call to a single NUMA node. Pages of the next call are distributed to another node chosen round-robin. Calling `mir_mem_pol_reset` resets the round.

14 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 explained in Appendix A. Contact MIR contributors for further 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 metrics 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
```

14.1 Enabling hardware performance counters

The recorder module can read hardware performance counters through PAPI during task switch events. Counter readings are attributed to the task that switched out and include effects of runtime system activity, system calls, interrupts, etc., that happened during task execution upto the event.

- 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 mir-recorder-trace-*.rec files created by the recorder during profiling. These files can processed using the rec2paraver.py script to obtain Paraver files as indicated earlier. They can also be processed using a special script for manual analysis.

```
$ $MIR_ROOT/scripts/profiling/thread/get-events.sh mir-recorder-trace.prv
$ sed -n '/EVENT/ {n;p}' mir-recorder-trace.pcf | cut -f 2,3
$ cat events-*-summary.csv
```

15 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 explained in Appendix B. Contact MIR contributors for further 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 *cutoff*=12.

Set the `--task-stats` flag to obtain per-task metrics 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 the run-independent unique identifier (called *lineage*), scatter, and source line number (definition sites) for tasks. Understand script 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. There is a special script called summarize-task-stats.R that summarizes the output of processing scripts (process-task-stats.R and `merge-task-stats.R`).

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

Processed task metrics in task-stats.processed can be visualized on grain graphs [1]. See the GitHub repository https://github.com/anamud/grain-graphs to understand how.

15.1 Profiling for-loop programs

Parallel for-loops are executed as special tasks by MIR. To enable per-chunk statistics, set `--chunks-are-tasks` in `MIR_CONF`. Process for-loop metrics by providing the `forloop` argument to task processing scripts.

```
$ MIR_CONF="--task-stats --chunks-are-tasks" <invoke MIR-linked for-loop program
    >
$ Rscript ${MIR_ROOT}/scripts/profiling/task/process-task-stats.R -d mir-task-stats
    -forloop
$ head loop-task-stats.processed
$ Rscript ${MIR_ROOT}/scripts/profiling/task/summarize-task-stats.R \
-d task-tats.processed -forloop
$ cat loop-task-stats.summarized
```

15.2 Merging task-based metrics

Merging task-based metrics from different sources into a common CSV file is beneficial for analysis. Let us look at a couple of examples of merging.

- Hardware performance counter readings collected by the recorder (Section 14) 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 multi-threaded execution using run-independent task identifiers. See the paper [1] for more details about the metric. 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*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
```

15.3 Instruction-level task profiler

MIR comes along with 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 operating system calls made within outline functions 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.
- Works in single-threaded mode only. Multi-threaded execution is not supported. This limitation only restricts profiling speed, and not the quality of the profiled data.

15.3.1 Build

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
```

15.3.2 Program preparation

The program to be profiled by the OFP should be compiled in a special manner using GCC to ensure outline functions of tasks are visible in object files.

Use an explicit two-step compilation process. First, compile source files into objects. Next, link the object files together to create the executable.

Provide the following flags during compilation: `-fno-inline-functions -fno-inline-functions-called-once -fno-optimize-sibling-calls -fno-omit-frame-pointer`. The SConstruct build file for example programs supplied with MIR uses these compilation flags to produce an executable with the suffix -prof.out.

15.3.3 Usage

The OFP is technically a *Pintool*, an inspection program created using Pin technology. Understand OFP options using the `-h` flag.

```
$ $PIN_ROOT/intel64/bin/pinbin -t $MIR_ROOT/scripts/profiling/task/obj-intel64/
    mir_of_profiler.so -h -- $MIR_ROOT/examples/OMP/fib-opt.out 1 1
...
-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]
...
```

Using the OFP 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 in a post-processing step.

Lets look at a demonstration of the above three steps for the Fibonacci example program prepared for profiling.

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 produces three lists as output. The lists are,

   - CHECKME_OUTLINE_FUNCTIONS: has names of task outline functions defined in the object files.
   - CHECKME_CALLED_FUNCTIONS: contains names of functions potentially called from inside the outline functions.
   - CHECKME_DYNAMICALLY_CALLED_FUNCTIONS: has 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 functions in CHECKME_OUTLINE_FUNCTIONS, duplicated/common list items, and empty lists. Usually, 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 the Fibonacci program is shown below.

   ```
   $ cd $MIR_ROOT/examples/OMP/fib
   $ $MIR_ROOT/scripts/profiling/task/of_finder.py ./fib-prof.out
   CHECKME_OUTLINE_FUNCTIONS=fib._omp_fn.0,fib._omp_fn.1,main._omp_fn
       .2,main._omp_fn.3
   CHECKME_CALLED_FUNCTIONS=inline_necessary,data_footprint_copy,...
   CHECKME_DYNAMICALLY_CALLED_FUNCTIONS=memcpy,pthread_attr_init
       ,...
   $ `$MIR_ROOT/scripts/profiling/task/of_finder.py -e ./fib-prof.out`
   ```

2. The next step is to profile the program and extract architecture independent metrics of tasks identified in the previous step. Profiling is performed by the OFP and and the MIR runtime system in tandem. Lets define a convenient shell function called `mir-inst-prof` that encapsulates profiling arguments to the OFP and the MIR runtime system.

   ```
   $ 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 the Fibonacci program.

```
$ mir-inst-prof -of $CHECKME_OUTLINE_FUNCTIONS -cf
      $CHECKME_CALLED_FUNCTIONS -df
      $CHECKME_DYNAMICALLY_CALLED_FUNCTIONS -- ./fib-prof.out 42
      12
$ ls
... mir-ofp-events ... mir-ofp-instructions ... 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 with other profiling data. The step simply involves executing a special merge script as shown below.

```
$ 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
$ head task-stats.merged
```

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

[3] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on NUMA systems," in *OpenMP in the Era of Low Power Devices and Accelerators.* Springer Berlin Heidelberg, 2013, pp. 156–170.

[4] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA'05. ACM, 2005, pp. 21–28.

[5] Muddukrishna, Ananya, P. A. Jonsson, and M. Brorsson, "Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors," *Scientific Programming*, no. 981759, 2015, reviewed and accepeted for publication by IOS publishers.

A Thread Profiling Data

The columns in CSV files containing thread profiling data are described in Table 1.

Name	**Description**
worker	Unique identifier of the worker
created	Number of tasks created
inlined	Number of tasks executed immediately
owned	Number of tasks executed from own queue
stolen	Number of tasks stolen from other queues
comm_tasks	Number of tasks with data footprints executed
total_comm_cost	Total communication cost of tasks with data footprints
avg_comm_cost	Average communication cost of tasks with data footprints
highest_comm_cost	Maximum communication cost among tasks with data footprints
lowest_comm_cost	Minimum communication cost among tasks with data footprints
comm_tasks_stolen_ ...by_diameter	Tasks with data footprints stolen from other queues sorted by NUMA distance

Table 1: Thread profiling data

B Task Profiling Data

The columns in CSV files containing task profiling data are described in Table 2.

Name	**Description**
task	Unique identifier

exec_cycles	Number of execution cycles
cpu_id	CPU that executed the task
create_instant	Instant created relative to parent execution start
exec_end_instant	Execution end instant relative to runtime system start
num_children	Number of children
creation_cycles	Number of cycles taken for creation
overhead_cycles	Number of cycles taken to create, schudule, and synchronize with children
work_cycles	Number of cycles taken for execution ignoring overheads. This is a measure of useful computation.
child_number	The nth child of the parent
wait_instants	Instants when waiting for children was initiated. This is relative to self execution start.
joins_at	Join point relative to parent
last_to_finish	True if last task executed on worker
leaf	True if task has no children
idle_join	The join point relative to the idle context (implicit task) that encountered the loop construct
lineage	Child number tracing back to root. This is a schedule-independent identifier.
tag	Name set by program or runtime system
source_line	Source line and file name of definition
outline_function	Outline function pointer
metadata	Useful extra information from program or runtime system
queue_size	Queue size when removed by worker. This is an alternative measure of parallelism.
parent	Parent identifier
parent_tag	Name of parent
parent_ind	Index of parent in tabular CSV data
parent_joins_at	Join point of parent
parent_overhead_cycles	Overhead cycles of parent
chunk_lineage	A tuple x-y, where x is the idle context join point, and y the start of the iteration range. This is a schedule-independent identifier for chunks.

chunk_work_balance	Maximum work cycles among chunk siblings divided by the mean. This is a measure of how evenly work is distributed across chunk siblings.
chunk_work_cpu	Chunk sibling work cycles across CPUs
chunk_work_cpu_balance	Chunk sibling work CPU balance across CPUs
inst_par_median	Median contribution to instantaneous parallelism by fragments
inst_par_min	Minimum contribution to instantaneous parallelism by fragments
inst_par_max	Maximum contribution to instantaneous parallelism by fragments
mem_hier_util	Ratio of cycles spent waiting for data to the number of work cycles. This is a measure of the locality exploited by the scheduler.
PAPI_*	PAPI event statistics
parallel_benefit	Ratio of work cycles to the overhead borne (overhead cycles) by parent
sync_cycles_per_child	Average number of cycles taken to synchronize with children
sibling_work_balance	Maximum work cycles among siblings divided by the mean
sibling_scatter	Median euclidean distance between identifiers of CPUs that executed siblings. This is a measure of locality exploited while executing siblings.
work_deviation	Difference in work cycles from another run
work_cycles_left	Work cycles for computing work deviation
work_cycles_right	Work cycles for computing work deviation

Table 2: Task profiling data