MIR User Guide

November 8, 2016

1 Introduction

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

- Detailed per-task performance profiling and visualization.
- Flexible, high performance task scheduling and data distribution policies. Examples: Locality-aware scheduling and data distribution on NUMA systems and work-stealing scheduling for multicore systems.
- Support for a capable subset of OpenMP 3.0 tasks and parallel for-loops.
- Competitive performance for medium-grained task-based programs.

2 Intended Audience

MIR is intended to be used by advanced task-based programmers. Knowledge of OpenMP compilation and role of runtime system in task-based programming 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 Installation

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

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

- A machine with x86 architecture.

- Linux kernel later than January 2012.
- GCC.
- Python (for executing scripts)
- GNU Binutils.
- Scons build system.
- Check, a unit testing framework.
- R (for executing scripts)
- These R packages:
  - data.table (for data structure transformations)

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

- Libraries libnuma and numactl (for data distribution and locality-aware scheduling on NUMA systems)
- GCC with OpenMP support (for linking task-based OpenMP programs)
- PAPI (for reading hardware performance counters during profiling)
- Paraver (for visualizing thread execution traces)
- Intel Pin sources (for profiling instructions executed by tasks)
- These R packages:
  - optparse (for parsing data)
  - igraph (for graph processing)
  - RColorBrewer (for colors)
  - ggplot2 and reshape2 (for plotting)
  - gdata, plyr, dplyr, scales (for data structure transformations)
  - pastecs (for summmarizing)
- yEd (for graph viewing, preferred)
- Cytoscape (for graph viewing)

3.1 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
```

3.2 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 reader.

3.3 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
```

3.3.1 Enabling data distribution and locality-aware scheduling on NUMA systems

- 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
```

3.3.2 Enabling OpenMP support

- Download the GPL implementation of the libGOMP interface from `https://github.com/anamud/mir-omp-int`. Point the environment variable MIR_OMP_INT_ROOT to the dowload 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
```

3.4 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
```

3.5 Examples

Run example programs in MIR_ROOT/examples.

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

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

4 Programming

4.1 OpenMP Interface

OpenMP support is restricted to the following interfaces:

- 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`

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

4.1.1 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.

4.2 Native Interface

The MIR library interface can be directly used to compose task-based programs. Look at mir_public_int.h in MIR_ROOT/src for interface details and example programs in MIR_ROOT/examples/native to understand usage.

4.3 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/or `CXXFLAGS`.

4.4 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
-s <str> (--schedule) task scheduling policy. Choose among 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!]
...
```

4.4.1 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
...
```

5 Profiling

MIR supports extensive and detailed thread-based and task-based profiling through scripts. Profiling data is stored mainly as CSV, Paraver, and GRAPHML files.

For CSV files, columns names are self-explanatory. Contact MIR contributors for clarifications about column names.

5.1 Thread-based Profiling

Thread states and events are the main performance indicators in thread-based profiling.

Enable 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
```

5.1.1 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. They 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
```

5.2 Task-based Profiling

Task are first-class citizens in task-based profiling.

Enable the `--task-stats` flag to collect task statistics in a CSV file called `mir-task-stats`. Inspect the file manually, or process it for summary information.

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

TODO: Explain file contents.

The processing step computes additional information such as number of tasks and task lineage (run-independent unique identifier for tasks). 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.

Hardware performance counter readings obtained during thread-based profiling can be summarized on a per-task basis.

```
$ $MIR_ROOT/scripts/profiling/thread/get-events-per-task.sh mir-recorder-
    trace-*.rec
$ cat accumulated-events.summary
$ cat accumulated-events.table
```

TODO: Explain file contents.

5.2.1 Instruction-level task profiling

MIR provides a Pin-based instruction profiler for tasks called the *Outline Function Profiler*. The profiler traces instructions executed within outline

functions of tasks in programs compiled with GCC. Instructions of dynamically linked functions and system calls called within the outline function are not traced. Read paper *Characterizing task-based OpenMP programs* (DOI: 10.1371/journal.pone.0123545) for more details.

Follow below steps to build and use the profiler.

- Get Intel Pin sources and set 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 the profiler.

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

- View profiler options using `-h`.

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

- The profiler requires handshaking with the runtime system. To enable handshaking, enable the `-p` flag in MIR_CONF.
- 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`.

  TODO: Explain file contents.

5.2.2 Visualization

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 -I 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.

5.3 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.