MIR User Guide

October 1, 2015

1 Introduction

MIR is an experimental task-based runtime system library written using C99. Prominent 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, work-stealing scheduling for multicore systems.
- Support for a capable subset of OpenMP 3.0 tasks and parallel for-loops interface.
- 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.

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.
- 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 task graph processing)
  - RColorBrewer (for colors)
  - ggplot2 and reshape2 (for plotting)
  - gdata, plyr, dplyr (for data structure transformations)
- yEd (for task graph viewing, preferred)
- Cytoscape (for task 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 of MIR is as follows :

```
. : 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 in MIR 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 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

- You should have received the GPL implementation of the libGOMP interface in a package separate from MIR sources. Point the environment variable MIR_OMP_INT_ROOT to the location of the package.
- Add 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
$ ln -s $MIR_OMP_INT_ROOT/mir_omp_int.h mir_omp_int.h
```

- Clean and rebuild MIR.

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

3.4 Testing

Run tests in MIR_ROOT/tests.

```
$ 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 example task-based programs 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])`. Can also be combined with `parallel` with applicable clauses.
- 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 tasks are supported by intercepting GCC translated calls to GNU libgomp. OpenMP 3.0 task interface support is therefore restricted to programs compiled using GCC.

4.1.1 Tips for writing MIR-supported OpenMP programs

- Use `taskwait` explicitly to synchronize tasks. Do not expect implicit taskwaits within thread barriers.
- Avoid distributing work to threads manually.
- 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 also 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 for interface usage examples.

4.3 Compiling and Linking

Add `-lmir-opt` to `LDFLAGS`. Enable MIR to intercept 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 which 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> for numa scheduling policy. Indicates data footprint size
    in bytes below which task is 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!]
--single-parallel-block
```

4.4.1 Binding workers to cores

MIR creates and binds one worker thread per core by default. Hardware threads are excluded 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.

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 load-balance information in a CSV file called `mir-worker-stats`.

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

TODO: Explain file contents.

MIR contains a `recorder` which produces execution traces. Use 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 thread. The files can be inspected individually or combined and visualized using Paraver.

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

Paraver configuration files for MIR traces are co-located with the script `rec2paraver.py`.

A set of `mir-recorder-state-time-*.rec` files are also created when `-r` is set. These files contain thread state duration information which can be accumulated for analysis without Paraver.

```
$ $MIR_ROOT/scripts/profiling/thread/get-states.sh \
mir-recorder-state-time
$ cat accumulated-state-file.info
```

TODO: Explain file contents.

5.1.1 Enabling hardware performance counters

MIR can read hardware performance counters through PAPI during task execution events. Events currently supported are the beginning and end of task execution. Hardware performance counters are not read during during a task switch 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`.

```
$ 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 values will appear in the `mir-recorder-trace-*.rec` files produced by the recorder during thread-based profiling. The counter readings can either be viewed on Paraver or accumulated for analysis outside Paraver.

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

TODO: Explain file contents.

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 plot and visualize the fork-join task graph.

```
$ MIR_CONF="--task-stats" <invoke mir-linked program>
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d mir-task-
    stats
```

TODO: Explain file contents.

The `mir-task-stats` file can be further processed for additional information such as number of tasks and task lineage (run-independent unique identifier for tasks). Processed information can also be used to visualize the 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. Note that performance counter readings will include the effects of all actions that occurred during task execution such as runtime system activity, system calls, interrupts etc.

```
$ $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 -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.

5.3 Profiling Case Study

TODO.