MIR User Guide

July 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. Example: Locality-aware scheduling and data distribution on NUMA systems, work-stealing scheduling for multicore systems.
- Supports for a capable subset of the OpenMP 3.0 tasks 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)
  - gdata, plyr, dplyr (for data structure transformations)
- yEd (for task graph viewing, preferred)
- Graphviz (for task graph viewing)
- 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
    |__helpers : helpful scripts, one-time hacks
    |__profiling : all things related to profiling
        |__task
        |__thread
|__programs : test programs, benchmarks
    |__common : build scripts
    |__native : native interface programs
        |__fib : Fibonacci program
            |__helpers : testing scripts
        |__bots : BOTS port
    |__omp : OpenMP interface programs
        |__fib
        |__bots
```

3.2 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
```

> Expert Tip: Ensure MIR_ROOT/src/SConstruct matches your build intention.

3.2.1 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 Testing

Try different runtime system configurations and program inputs on Fibonacci in MIR_ROOT/programs/native/fib. All programs under MIR_ROOT/programs can be used for testing.

```
$ cd $MIR_ROOT/programs/native/fib
$ scons -c
$ scons
$ ./fib-verbose
$ ./fib-debug
$ ./fib-opt
```

> Note: A dedicated test suite will be added soon, so watch out for that!

4 Programming

4.1 OpenMP 3.0 Tasks Interface

A restricted subset of OpenMP 3.0 tasks — the `task` and `taskwait` constructs — is supported. Although minimal, the subset is sufficient for writing most task-based programs.

> 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

- 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 data environment.
  - Use `taskwait` to synchronize tasks.
  - Use taskwait explicitly. Do not expect implicit taskwaits within thread barriers.
- 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.
  - Do not use the `parallel` construct to share work.
  - Do not use barriers to synchronize threads. Do not expect implicit barriers at the end parallel blocks.
- Use GCC atomic builtins for flushing and atomic operations.
- Study example programs in MIR_ROOT/programs/omp.

A simple set of steps for producing MIR-supported OpenMP programs is given below:

1. When parallel execution is required, create a `parallel` block followed immediately by a `single` block.
2. Use the `task` construct within the `single` block to parallelize work.
3. Synchronize tasks using the `taskwait` construct explicitly. Do not rely on implicit barriers and taskwaits.
4. It is strongly recommended to parallelize work inside a master task context. This simplifies interpreting MIR profiling results.
5. Compile and link with the native GCC OpenMP implementation. Ensure the program runs correctly.
6. Ensure that `parallel` block do not share work and that all tasks are synchronized explicitly.
7. Compile and link with the appropriate MIR library (opt/debug). The program is now ready.

An example program built using above steps follows.

```
int main(int argc, char *argv[])
{
#pragma omp parallel
{
#pragma omp single
{
// Create master task
#pragma omp task
{
    // Now parallelize the work.
    // For example, lets say the work is to
    // ... create a 1000 parallel instances of the fuction foo.
    for(int i=0; i<1000; i++)
    {
        #pragma omp task firstprivate(i)
            foo(i);
    }

    // Wait for tasks to finish
    #pragma omp taskwait
}
// Wait for master task to finish
#pragma omp taskwait
} // omp single end
} // omp parallel end

    return 0;
}
```

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 programs in MIR_ROOT/programs/native for interface usage examples. A simple program using the native interface is shown below.

```
#include "mir_public_int.h"
void foo(int id)
{
    printf(stderr, "Hello from task %d\n", id);
}

// Outline function for foo.
struct foo_of_arg_t
```

```
{
    int id;
};
void foo_of(void* arg)
{
    struct foo_of_arg_t* farg = (struct foo_of_arg_t*)(arg);
    foo(farg->id);
}

// The master task
// Having a master task helps to interpret profiling results
void master_task()
{
    // Create a 1000 instance of foo.
    for(int i=0; i<1000; i++)
    {
        struct foo_of_arg_t arg;
        arg.id = i;
        mir_task_create((mir_tfunc_t) foo_of,
                        &arg,
                        sizeof(struct foo_of_arg_t),
                        0, NULL, NULL);
    }

    // Wait for tasks to finish
    mir_task_wait();
}

// Outline function for the master task
void master_task_of(void* arg)
{
    master_task();
}

int main(int argc, char *argv[])
{
    // Initialize the runtime system
    mir_create();

    // Create master task
    mir_task_create((mir_tfunc_t) master_task_of,
                    NULL,
                    0,
                    0, NULL, NULL);

    // Wait for master task to finish
    mir_task_wait();

    // Release runtime system resources
```

```
    mir_destroy();

    return 0;
}
```

4.3 Compiling and Linking

Add `-lmir-opt` to `LDFLAGS`. Enable MIR to intercept 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.

```
$ cd $MIR_ROOT/test/fib
$ scons
$ MIR_CONF="-h" ./fib-opt 3
MIR_INFO: Valid options in MIR_CONF environment variable ...
-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!]
```

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"
$ cd $MIR_ROOT/programs/native/fib
$ scons
$ ./fib-debug 10 3
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" ./fib-opt
$ 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" ./fib-opt
$ $MIR_ROOT/scripts/profiling/thread/rec2paraver.py \
mir-recorder-trace-config.rec
$ wxparaver mir-recorder-trace.prv
```

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" ./fib-opt
$ 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 mir-task-stats.info
$ head mir-task-stats.lineage
$ head mir-task-stats.processed
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d mir-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 occured 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 (csv) from outline functions
  -pr output file prefix [default mir-ofp]
  ...
  ```

The profiler requires outline function names under the argument `-of`. The argument `-cf` accepts names of functions which are called within tasks. The argument `--` separates profiled program invocation from profiler arguments.

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

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

The profiler produces following outputs:

- Per-task instructions in a CSV file called `mir-ofp-instructions`. Example contents of the file are shown below. TODO: Add file contents. Each line shows instruction and code properties of a distinct task executed by the program. Properties are described below.
  - task: Identifier of the task.
  - ins_count: Total number of instructions executed by the task excluding instructions of system calls, dynamically linked functions and runtime system functions.
  - stack_read: Number of read accesses to the stack while executing instructions.
  - stack_write: Number of write accesses to the stack while executing instructions.
  - ccr: Computation to Communication Ratio. Indicates number of instructions executed per read or write access to memory.
  - clr: Computation to Load Ratio. Indicates number of instructions executed per read access to memory.

- mem_read: Number of read accesses to memory (excluding stack) while executing instructions.
- mem_write: Number of write accesses to memory (excluding stack) while executing instructions.
- outl_func: Name of the outline function of the task.

- Per-task events in a file called `mir-ofp-events`. Example contents of the file are shown below.

```
task,ins_count,[create],[wait]
14,446,[],[]
15,278,[],[]
10,60,[32,43,],[47,]
```

  Each line in the file shows events for a distinct task executed by the program. Event occurance is indicated in terms of instruction count. Events currently supported are:

  - `create`: Indicates when child tasks were created. Example: [32,43] indicates the task 10 created its first child at instruction 32 and second child at 43. Tasks 14 and 15 did not create children tasks.
  - `wait`: Indicates when child tasks were synchronized. Example: [47,] indicates the task 10 synchronized with all children created prior at instruction 47.

- Program memory map in a file called `mir-ofp-mem-map`. This is a copy of the memory map file of the program from the /proc filesystem.

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 Graphviz, 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 mir-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`. Huge graphs with 50000+ tasks take a long time to plot. To save time, plot the task graph as a tree using option `--tree`.

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 mir-
    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: Fibonacci

The Fibonacci program is found in MIR_ROOT/programs/native/fib. The program takes two arguments – the number n and the depth cutoff for recursive task creation. Let us see how to profile the program for task-based performance information.

Compile the program for profiling – remove aggressive optimizations and disable inlining so that outline functions representing tasks are visible to the Pin-based instruction profiler. Running scons in the program directory builds the profiler-friendly executable called `fib-prof`.

```
$ cd $MIR_ROOT/programs/native/fib
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: debug-build opt-build prof-build
    verbose-build
...
gcc -o prof-build/fib.o -c -std=c99 -Wall -Werror -Wno-unused-function -
    Wno-unused-variable -Wno-unused-but-set-variable -Wno-maybe-
    uninitialized -fopenmp -DLINUX -I/home/ananya/mir-dev/src -I/home/
    ananya/mir-dev/programs/common -O2 -DNDEBUG -fno-inline-functions
    -fno-inline-functions-called-once -fno-optimize-sibling-calls -fno-omit
    -frame-pointer -g fib.c
```

```
...
gcc −o fib−prof prof−build/fib.o −L/home/ananya/mir−dev/src −lpthread −lm −
    lmir−opt
```

> Tip: Look at the `SConstruct` file in MIR_ROOT/test/fib and build output to understand how the profiling-friendly build is done.

Identify outline functions and functions called within tasks of the `fib-prof` program using the script `of_finder.py`. The script searches for known outline function name patterns within the object files of `fib-prof`. The script lists outline functions as `CHECKME_OUTLINE_FUNCTIONS` and all function symbols within the object files as `CHECKME_CALLED_FUNCTIONS`.

```
$ cd $MIR_ROOT/programs/native/fib
$ $MIR_ROOT/scripts/profiling/task/of_finder.py −v prof−build/∗.o
Using "._omp_fn.|ol_" as outline function name pattern
Processing file: prof−build/fib.o
CHECKME_OUTLINE_FUNCTIONS=ol_fib_0,ol_fib_1,ol_fib_2
CHECKME_CALLED_FUNCTIONS=fib_seq,fib,get_usecs,main
```

> Expert Tip: Ensure that functions listed by CHECKME_OUTLINE_FUNCTIONS are those generated by GCC. Inspect the abstract syntax tree (use compilation option `-fdump-tree-optimized`) and source files.

The functions in the `CHECKME_CALLED_FUNCTIONS` list should be treated as functions potentially called within task contexts. Inspect program sources and exclude those which are not called within tasks. By looking at Fibonacci program sources, we can exclude `main` and `get_usecs` from the called function list in `CHECKME_CALLED_FUNCTIONS`.

> Tip: If in doubt or when sources are not available, use the entire `CHECKME_CALLED_FUNCTIONS` list.
>
> Expert Tip: Identifying functions called by tasks is necessary because the instruction count of these functions are added to the calling task's instruction count.

Start the instruction profiler with appropriate arguments to profile `fib-prof`. Also collect task statistics at the same time.

```
$ mir-inst-prof \
   -of ol_fib_0,ol_fib_1,ol_fib_2 \
   -cf fib,fib_seq \
   -- ./fib-prof 10 4
```

> Tip: If you plan to use the entire `CHECKME_CALLED_FUNCTIONS` and `CHECKME_OUTLINE_FUNCTIONS` lists, then you can use them as arguments to **`mir-inst-prof`** in the following manner:
>
> ```
> $ `$MIR_ROOT/scripts/profiling/task/of_finder.py -e prof-build
>     /*.o`
> $ mir-inst-prof \
>    -of $CHECKME_OUTLINE_FUNCTIONS
>    -cf $CHECKME_CALLED_FUNCTIONS
>    -- ./fib-prof 10 4
> ```
>
> The `-e` option of `of_finder.py` outputs outline and called function lists in the BASH export format. The backticks evaluate the output as commands.

Inspect instruction profiler output.

```
$ head mir-ofp-instructions
$ head mir-ofp-events
```

Inspect task statistics.

```
$ head mir-task-stats
```

Summarize task statistics.

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

Combine the instruction-level information produced by the instruction profiler with the task statistics produced by the runtime system into a single CSV file. Note that these files come from the same run.

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

Plot task graph using combined performance information and view on the yEd graph viewer.

```
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d mir-task-
    perf -p color
$ yed task-graph.graphml
```

Tip: Import task graph property mapping settings in MIR_ROOT/scripts/profiling/task/yed-mir-task-graph-settings.cnfx into yEd.

TODO: Demonstrate automatic analysis of task performance problems using `analyze` option of the task graph plotter.