MIR User Guide

Introduction

MIR is a task-based runtime system library written using C99. MIR scales well for medium-grained task-based programs. MIR provides a simple native interface for writing task-based programs. In addition, a subset of the OpenMP 3.0 tasks interface is supported. MIR is flexible --- the user can experiment with different scheduling policies. Example: Locality-aware scheduling and data distribution on NUMA systems. MIR supports extensive performance analysis and profiling features. Users can quickly solve performance problems using detailed thread-based and task-based performance information profiled by MIR.

Intended Audience

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

Installation

Mandatory Requirements

- Machine with x86 architecture.
- Linux kernel later than January 2012.
- GCC.
- Binutils.
- Scons build system.

Optional Requirements

Enabling extended features such as profiling, locality-aware scheduling and data distribution requires:

- 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)
- Python 2.X and 3.X (for executing profiling scripts)
- Intel Pin sources (for profiling instructions executed by tasks)
- R (for executing profiling scripts)
- R packages:
  - optparse (for parsing data)
  - igraph (for task graph processing)
  - RColorBrewer (for colors)
  - gdata, plyr, dplyr, data.table (for data structure transformations)
- Graphviz (for task graph plotting)

Source Structure

The source repository is structured intuitively. Files and directories have purpose-oriented names.

```
. : MIR_ROOT
|__docs : documentation
```

```
|__src : runtime system sources
    |__scheduling : scheduling policies
    |__arch : architecture specific sources
|__scripts
    |__helpers : helpful scripts, dirty hacks
    |__profiling : all things related to profiling
        |__task
        |__thread
|__programs : test programs, benchmarks
    |__common : build scripts
    |__native : native interface programs
        |__fib
            |__helpers : testing scripts
    |__bots : BOTS port
    |__omp : OpenMP interface programs
        |__fib
```

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 this to .bashrc to avoid repeated initialization.

- Build.

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

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

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
```

Testing

The Fibonacci program in MIR_ROOT/programs/native/fib is recommended for testing. Try different runtime system configurations and program inputs. Verify correctness and scalability. Other programs in MIR_ROOT/programs can also 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!

Programming

Native Interface

The native interface for task-based programming is friendly, even to non-experts. Look at mir_public_int.h in MIR_ROOT/src for interface details and programs in MIR_ROOT/programs 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);
}

// Task outline function argument
struct foo_wrapper_arg_t
{
int id;
};

// Task outline function
void foo_wrapper(void* arg)
{
    struct foo_wrapper_arg_t* farg = (struct foo_wrapper_arg_t*)(arg);
    foo(farg->id);
}

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

    // Create as many tasks as there are threads
    int num_workers = mir_get_num_threads();
    for(int i=0; i<num_workers; i++)
    {
        struct foo_wrapper_arg_t arg;
        arg.id = i;
        mir_task_create((mir_tfunc_t) foo_wrapper,
                        &arg,
                        sizeof(struct foo_wrapper_arg_t),
                        0, NULL, NULL);
    }

    // Wait for tasks to finish
    mir_task_wait();

    // Release runtime system resources
    mir_destroy();

    return 0;
}
```

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.

The `parallel` construct is deprecated. A team of threads is created when `mir_create` is called. The team is disbanded when `mir_destroy` is called.

> 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.

Tips for writing MIR-supported OpenMP programs

- Initialize and release the runtime system explicitly by calling `mir_create` and `mir_destroy`.
- Do not think in terms of threads.
    - Do not use the `parallel` construct to share work.
    - Do not use barriers to synchronize threads.
- Think solely in terms of 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 `mir_lock` instead of the `critical` construct or use OS locks such as `pthread_lock`.
- 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:

i. When parallel execution is required, create a `parallel` block with `default(none)` followed immediately by a `single` block. The `default(none)` clause avoids incorrect execution due to assumed sharing rules.

ii. Use the `task` construct within the `single` block to parallelize work.

iii. Synchronize tasks using the `taskwait` construct explicitly. Do not rely on implicit barriers and taskwaits.

iv. Parallelizing work inside a master task context is helpful while interpreting profiling results.

v. Compile and link with the native OpenMP implementation (preferably libgomp) and check if the program runs correctly.

vi. Comment out the `parallel` and `single` blocks, initialize the MIR runtime system right in the beginning of the program by calling `mir_create` and release it at the end of the program by calling `mir_destroy`, include `mir_public_int.h`.

vii. Compile and link with the appropriate MIR library (opt/debug). The program is now ready.

The native interface example rewritten using above steps is shown below.

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

//#pragma omp parallel default(none)
//{
//#pragma omp single
//{
// Master task context: helpful for interpreting profiling results.
#pragma omp task
{
    // Now parallelize the work involved
    // Work in this case: create as many tasks
    // ... as there are threads
    int num_workers = mir_get_num_threads();
    for(int i=0; i<num_workers; 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
//}
//}
    // Release runtime system resources
    mir_destroy();

    return 0;
}
```

Compiling and Linking

Look at `SConstruct`, the Scons build file accompanying each program to understand how to compile and link with the MIR library. Observing verbose build messages is also recommended.

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 print this help message
-w=<int> number of workers
-s=<str> task scheduling policy. Choose among central, central-stack, ws, ws-de and numa.
-r enable worker recorder
-x=<int> task inlining limit based on num tasks per worker
-i collect worker statistics
-l=<int> worker stack size in MB
-q=<int> task queue capacity
-m=<str> memory allocation policy. Choose among coarse, fine and system.
-y=<csv> schedule policy specific parameters. Policy numa: data size in bytes below which task is dealt to worker's private queue.
-g collect task statistics
-p enable handshake with Outline Function Profiler [Note: Supported only for a single worker!]
```

Binding workers to cores

MIR creates and binds one worker thread per core (including hardware threads) by default. 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
...
```

Profiling

MIR supports extensive thread-based and task-based profiling.

Thread-based Profiling

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

Enable the `-i` flag to get basic load-balance information in a CSV file called `mir-worker-stats`.

```
$ MIR_CONF="-i" ./fib-opt
$ cat mir-worker-stats
```

TODO: Explain file contents

MIR contains a module called the `recorder` which produces detailed 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
```

Enabling hardware performance counters

MIR can read hardware performance counters during thread events. This process is not fully automated and needs a little bit of hands-on work from the user.
* 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 the preprocessor definition `MIR_RECORDER_USE_HW_PERF_COUNTERS` in `MIR_ROOT/src/mir_defines.h`.

```
$grep -i HW_PERF $MIR_ROOT/src/mir_defines.h
#define MIR_RECORDER_USE_HW_PERF_COUNTERS
```

- Enable PAPI hardware performance counters of interest in `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 readings will be now be added to `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
```

Task-based Profiling

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

Enable the `-g` 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="-g" ./fib-opt
$ Rscript ${MIR_ROOT}/scripts/profiling/task/plot-task-graph.R -d mir-task-stats -c color
```

TODO: Explain file contents.

The `mir-task-stats` can also be processed for additional information such as number of tasks and task lineage (UID for tasks).

```
$ Rscript ${MIR_ROOT}/scripts/profiling/task/process-task-stats.R mir-task-stats
$ cat mir-task-stats.info
num_tasks: 15
joins_at_summary: 1 2 2 1.875 2 2
$ cat mir-task-stats.lineage
"task","parent","lineage"
1,0,"0-1"
2,1,"0-1-2"
3,1,"0-1-3"
...
```

Instruction-level task profiling

MIR provides a Pin-based instruction profiler that traces instructions executed by tasks. Technically, the profiler traces instructions executed within outline functions of tasks in programs compiled using GCC. 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
```

- 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
...
-c  [default ]
specify functions called (csv) from outline functions
-o  [default mir-ofp]
specify output file suffix
-s  [default ]
specify outline functions (csv)
...
```

The profiler requires outline function names under the argument `-s`. The argument `-c` 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.

- Create a handy alias for invoking the profiler.

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

- The profiler produces following outputs:

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

```
"task","parent","joins_at","child_number","num_children","core_id","exec_cycles","ins_count","stack_read","stack_write","mem_fp","ccr","clr","mem_read","mem_write","outl_func"
1,0,0,0,2,0,21887625,58,10,15,5,12,15,4,1,"ol_fib_2"
2,1,0,1,2,0,610035,60,10,15,5,12,15,4,1,"ol_fib_0"
```

```
3,1,0,2,2,0,3183115,60,10,15,5,12,15,4,1,"ol_fib_1"
```

Each line shows instruction and code properties of a distinct task executed by the program. Properties are described below.

- `task`: Identifier of the task.
- `parent`: Identifier of the parent task.
- `joins_at`: The order of synchronizing with the parent task context.
- `child_number`: Order of task creation by parent.
- `num_children`: Indicates the number of child tasks created by the task.
- `exec_cycles`: Number of cycles spent executing the task including child task creation and synchronization.
- `core_id`: Identifier of the core that executed the task.
- `ins_count`: Total number of instructions executed by the task.
- `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.

ii. 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.

iii. 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.

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.

- 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 -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 `–noCPE`.

- Huge graphs with 50000+ tasks take a long time to plot. Plot the task graph as a tree to save time.

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

- 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 into a single CSV file. Plot task graph using combined performance information.

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

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 `OUTLINE_FUNCTIONS` and all function symbols within the object files as `CALLED_FUNCTIONS`.

```
$ cd $MIR_ROOT/programs/native/fib
$ $MIR_ROOT/scripts/profiling/task/of_finder.py prof-build/*.o
Using "._omp_fn.|ol_" as outline function name pattern
Processing file: prof-build/fib.o
OUTLINE_FUNCTIONS=ol_fib_0,ol_fib_1,ol_fib_2
```

```
CALLED_FUNCTIONS=fib_seq,fib,get_usecs,main
```

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

The functions in the `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 `CALLED_FUNCTIONS`.

> Tip: If in doubt or when sources are not available, use the entire `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`.

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

> Tip:
> If you get a missing link-library error, add PIN_ROOT/intel64/runtime to LD_LIBRARY_PATH.

- Inspect instruction profiler output.

```
$ head mir-ofp-instructions
"task","parent","joins_at","child_number","num_children","core_id","exec_cycles","ins_count","stack_read","stack_write"
,"mem_fp","ccr","clr","mem_read","mem_write","outl_func"
1,0,0,0,2,0,21887625,58,10,15,5,12,15,4,1,"ol_fib_2"
2,1,0,1,2,0,610035,60,10,15,5,12,15,4,1,"ol_fib_0"
3,1,0,2,2,0,3183115,60,10,15,5,12,15,4,1,"ol_fib_1"
...
$ head mir-ofp-events
task,ins_count,[create],[wait]
14,446,[],[]
15,278,[],[]
10,60,[32,43,],[47,]
...
```

- Collect task statistics.

```
MIR_CONF="-g" ./fib-prof 10 4
```

> Tip:
> Generate task statistics information simultaneously with other statistics to maintain consistency.

- Summarize task statistics.

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

```
num_tasks: 15
joins_at_summary: 1 2 2 1.875 2 2
```

- Combine the instruction-level information produced by the instruction profiler with the task statistics produced by the runtime system into a single CSV file.

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

- Plot task graph using combined performance information and view on YEd.

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