Annotated Task Graph (ATG)

Ananya Muddukrishna

ananya@kth.se

1 Introduction

The Annotated task Graph (ATG) refers to information obtained by the instruction-level profiling feature of the MIR runtime system library. Instruction-level profiling is performed using a custom Pin call-graph profiling tool. Note that instruction-level task profiling is restricted to programs compiled using GCC.

The ATG is available in two forms - raw and visual.

2 Getting the ATG

Let us obtain the ATG for the fib program in MIR_ROOT/test/fib. The fib program takes two arguments. The first is the number n and the second is the depth cutoff for recursive task creation.

We first have to compile the fib program without aggressive optimizations and disable inlining so that outline functions representing tasks are visible to Pin call-graph profiler. Look at the SConstruct file in MIR_ROOT/test/fib and build output to understand which arguments are supplied to the compiler to get the profiling-specialized executable called fib-prof. Build output pertaining to fib-prof are show below.

```
$ cd $MIR_ROOT/test/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/test/common -O1 -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
...
scons: done building targets.
```

We next identify outline functions and functions called within tasks of the fib program. To do so, we use a simple script - profiler_params.py - which searches for known outline function name patterns within the object files of the program fib-prof. The script lists outline functions names as OUTLINE_FUNCTIONS and all function symbols within the object files as CALLABLE_FUNCTIONS. Ensure that OUTLINE_FUNCTIONS listed are those generated by GCC by inspecting the asbtract syntax tree (use compilation option -fdumptreeoptimized) and source files or verifying with a local OpenMP expert. We treat CALLABLE_FUNCTIONS as potentially callable functions and filter out those which are certianly called by tasks defined in the program. Inspecting program source files is a quick way to filtering out certainly called functions. By looking at fib program sources, we can exclude main and get_usecs from CALLABLE_FUNCTIONS. If in doubt or when sources are not available, use the entire list. Identifying functions called by tasks is necessary because the instruction count of these functions are added to the calling task's instruction count. Note down OUTLINE_FUNCTIONS and those functions names you have retained in CALLABLE_FUNCTIONS.

```
$ cd $MIR_ROOT/test/fib

$ echo "Examining executable for names of outline and callable functions ..."
$ $MIR_ROOT/scripts/task-graph/profiler_params.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
CALLABLE_FUNCTIONS=fib_seq,fib,get_usecs,main
```

Now we invoke the Pin call-graph profiler - mir_outline_function_profiler.so - with appropriate arguments to profile fib-prof and obtain instruction-level information for tasks. The profiler accepts outline function names under the argument -s and names of function which are called within tasks under the argument -c. Use these arguments to indicate the functions you have noted

down using profiler_params.py. The -o argument is used as prefix to file names produced during profiling. The −− argument indicates the profiled program and its arguments, which in our case is fib-prof with inputs n=10 and cutoff=4. Use the -h argument for help information about the profiler. We additionally instruct MIR to provide fork-join task graph information using MIR_CONF="-w=1 -g -p" which is mandatory and constant for building the ATG. The MIR_CONF argument -w=1 enables single-threaded execution, -g enables fork-join task graph building and -p enables hand-shaking between the MIR runtime system and the Pin call-graph profiler. Use the -h argument for help information about MIR_CONF.

```
$ LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PIN_ROOT/intel64/runtime \
    MIR_CONF="-w=1 -g -p" \
    $PIN_ROOT/intel64/bin/pinbin \
    -t ${MIR_ROOT}/scripts/task-graph/obj-intel64/
      mir_outline_function_profiler.so \
    -o fib_test \
    -s ol_fib_0,ol_fib_1,ol_fib_2 \
    -c fib,fib_seq \
    -- ./fib-prof 10 4

$ mv mir-task-graph fib_test-fork_join_task_graph
```

Now, we can get some basic task-based information by processing the fork-join task graph produced by MIR.

```
$ echo "Summarizing fork-join task graph ..."
$ Rscript ${MIR_ROOT}/scripts/task-graph/mir-fork-join-graph-info.R
      fib_test-fork_join_task_graph
```

We can also plot the fork-join task graph.

```
$ echo "Plotting fork-join task graph ..."
$ Rscript ${MIR_ROOT}/scripts/task-graph/mir-fork-join-graph-plot.R
      fib_test-fork_join_task_graph color
```

Huge graphs (50000+ tasks) take a long time to plot. We can plot the fork-join task graph as a tree to save time.

```
$ echo "Plotting fork-join task graph ..."
$ Rscript ${MIR_ROOT}/scripts/task-graph/mir-tree-graph-plot.R fib_test-
      fork_join_task_graph color
```

Now, we combine the instruction-level profile produced by the Pin call-graph profiler with the fork-join task graph information produced by MIR. We

call this process as annotating the fork-join task graph with instruction-level information. The annotation process completes production of the raw format of the ATG.

```
$ echo "Annotating fork-join task graph ..."
$ Rscript ${MIR_ROOT}/scripts/task-graph/mir-annotate-graph.R fib_test-
    fork_join_task_graph fib_test-call_graph fib_test
```

We can use raw ATG information to plot the visual form of the ATG.

```
$ echo "Plotting annotated task graph ..."
$ Rscript ${MIR_ROOT}/scripts/task-graph/mir-annotated-graph-plot.R
    fib_test-annotated_task_graph color
```

Let us list the files produced by running the above commands. See Table 1 for a description of files produced.

```
$ echo "Listing ATG files ..."
$ ls fib_test*
```

3 Raw format of the ATG

The ATG raw format consists of per-task properties and fine-grained timing information.

3.1 Properties

Properties are indicated in a file with the suffix *annotated_task_graph*. An example is shown below.

```
$ head fib_test-annotated_task_graph
"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","name"
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"
4,3,0,1,2,0,38565,60,10,15,5,12,15,4,1,"ol_fib_0"
5,3,0,2,2,0,7192500,60,10,15,5,12,15,4,1,"ol_fib_1"
6,5,0,1,0,0,4710295,173,40,40,5,35,43,4,1,"ol_fib_0"
7,5,0,2,0,0,12847265,110,25,25,5,22,28,4,1,"ol_fib_1"
8,4,0,1,0,0,15955,278,65,65,5,56,70,4,1,"ol_fib_0"
9,4,0,2,0,0,26420,173,40,40,5,35,43,4,1,"ol_fib_1"
```

File name	Description
fib_test-call_graph	Instruction-level information of tasks
fib_test-mem_map	Memory map of program execution
fib_test-fork_join_task_graph	Parent-child relationship and execution information for tasks such as identification of executing cores and execution time
fib_test-create_wait_instants	Instruction instants at which tasks were created and synchronized
fib_test-annotated_task_graph	Raw format of the ATG combining instruction-level and parent-child information
fib_test-annotated_task_graph.adjm	Adjacent matrix representation of the visual format of ATG
fib_test-annotated_task_graph.dot	Dot representation of the visual format of ATG
fib_test-annotated_task_graph.edgelist	Edgelist representation of the visual format of ATG
fib_test-annotated_task_graph.graphml	GraphML representation of the visual format of ATG
fib_test-annotated_task_graph.info	Summary information about visual format of the ATG. Includes work, span and critical path from Cilk theory.
fib_test-fork_join_task_graph.adjm	Adjacent matrix representation of the visual format of ATG without instruction-level information
fib_test-fork_join_task_graph.dot	Dot representation of the visual format of ATG without instruction-level information
fib_test-fork_join_task_graph.edgelist	Edgelist representation of the visual format of ATG without instruction-level information
fib_test-fork_join_task_graph.graphml	GraphML representation of the visual format of ATG without instruction-level information
fib_test-fork_join_task_graph.info	Summary information about visual format of ATG without instruction-level information. Includes number of tasks and join degree distribution.

Table 1: ATG files

Each line shows of the property file shows properties of an explicit task executed by the program. The first line, or the header, shows names of the

properties. Properties are also called annotations. See Table 2 for property descriptions.

Field	Description
task	Identifier of the task
parent	Identifier of the parent task of the task
joins_at	Indicates at which call to taskwait in the parent the task synchronized. Example: 0 indicates the task synchronized with the first call to taskwait in the parent. Several children can synchronize at the same call.
child_number	Indicates the order of task creation by parent
num_children	Indicates the number of child tasks created by the task
exec_cycles	Indicates the number of cycles spent executing the task including child task creation and synchronization
core_id	Indicates which core executed the task
ins_count	Indicates total number of instructions executed by the task. Profiling parameters indicate which instructions to count. Typically, instructions part of runtime system calls are excluded and calls to statically-linked functions are included.
stack_read	Indicates number of read accesses to the stack while executing instructions
stack_write	Indicates 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	Indicates number of read accesses to memory (excluding stack) while executing instructions
mem_write	Indicates number of write accesses to memory (excluding stack) while executing instructions
name	Indicates name of the outline function of the task

Table 2: Properties

3.2 Fine-grained timing information

Fine-grained timing information is provided in a file with suffix *create_ wait_ instants*. An example is shown below.

```
$ head -20 fib_test-create_wait_instants
task,ins_count,[create],[wait]
14,446,[],[]
```

```
15,278,[],[]
10,60,[32,43,],[47,]
12,278,[],[]
13,173,[],[]
11,60,[32,43,],[47,]
2,60,[32,43,],[47,]
8,278,[],[]
9,173,[],[]
4,60,[32,43,],[47,]
6,173,[],[]
7,110,[],[]
5,60,[32,43,],[47,]
3,60,[32,43,],[47,]
1,58,[30,41,],[45,]
```

Each line shows timing information of an explicit task executed by the program. Timing is indicated in terms of instruction count. The first line contains header information. See Table 3 for header field descriptions.

Field	**Description**
task	Identifier of the task
ins_count	Indicates total number of instructions executed by the task. Profiling parameters indicate which instructions to count. Typically, instructions part of runtime system calls are excluded and calls to statically-linked functions are included.
create	Indicates when child tasks were created. For example, [32,43] indicates the task created its first child at instruction count 32 and second child at 43.
wait	Indicates when child tasks were synchronized. For example, [47,] indicates the task synchronized with all children created prior at instruction count 47.

Table 3: Timing

4 Visual format of the ATG

The visual format gives shape to the raw format of the ATG. It describes task-based execution in an intuitive manner allowing the programmer to spot performance problems. The visual format can be viewed using graph visualization tools such as dot, yEd and cytoscape. See Figure 1 for visualization of the ATG on yEd and Figure 2 for visualization of the ATG on Dot. Details of the visual format are subject of a scientific paper under review and will be made available soon.

```
$ echo "Visualizing annotated task graph ..."
$ dot -Tps fib_test-annotated_task_graph.dot >  fib_test-
    annotated_task_graph.dot.ps
$ yed fib_test-annotated_task_graph.graphml
```

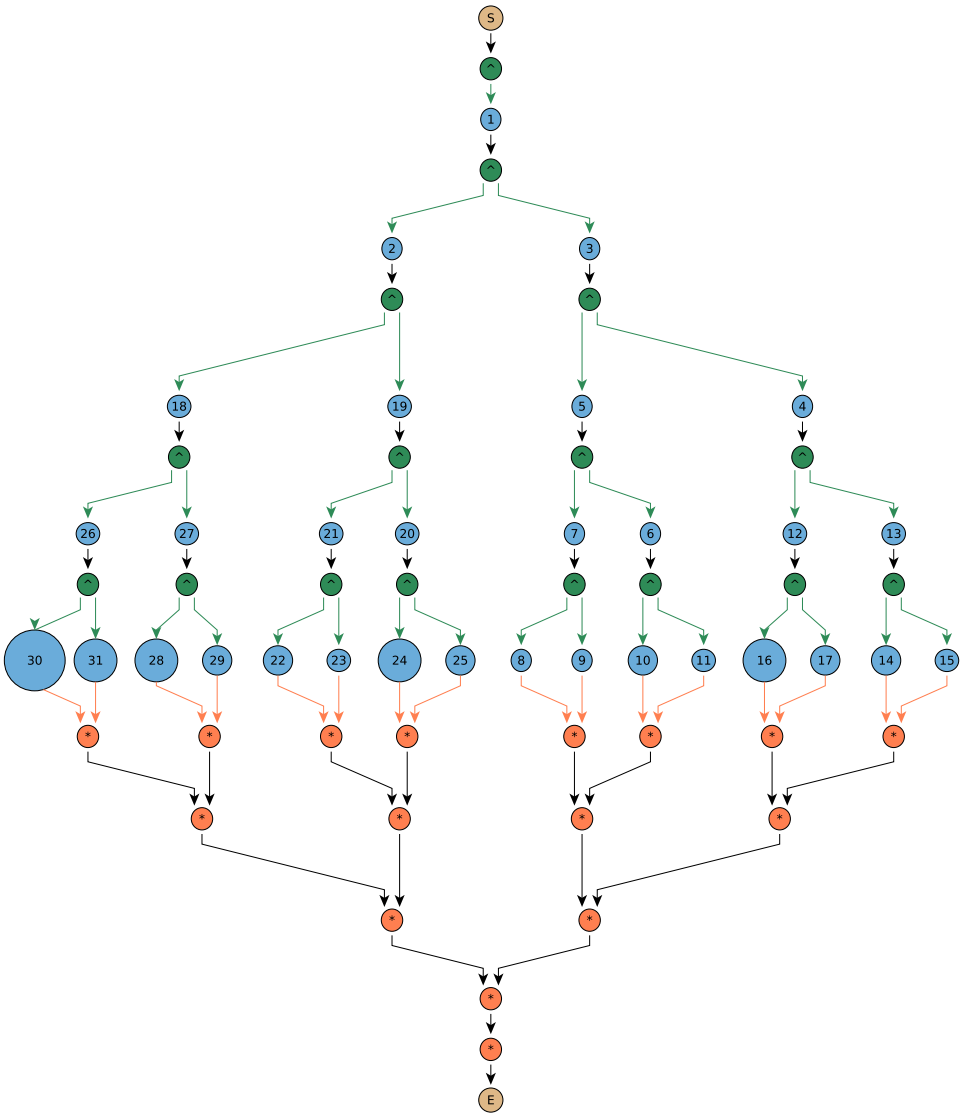

Figure 1: fib_test-annotated_task_graph.graphml viewed on yEd

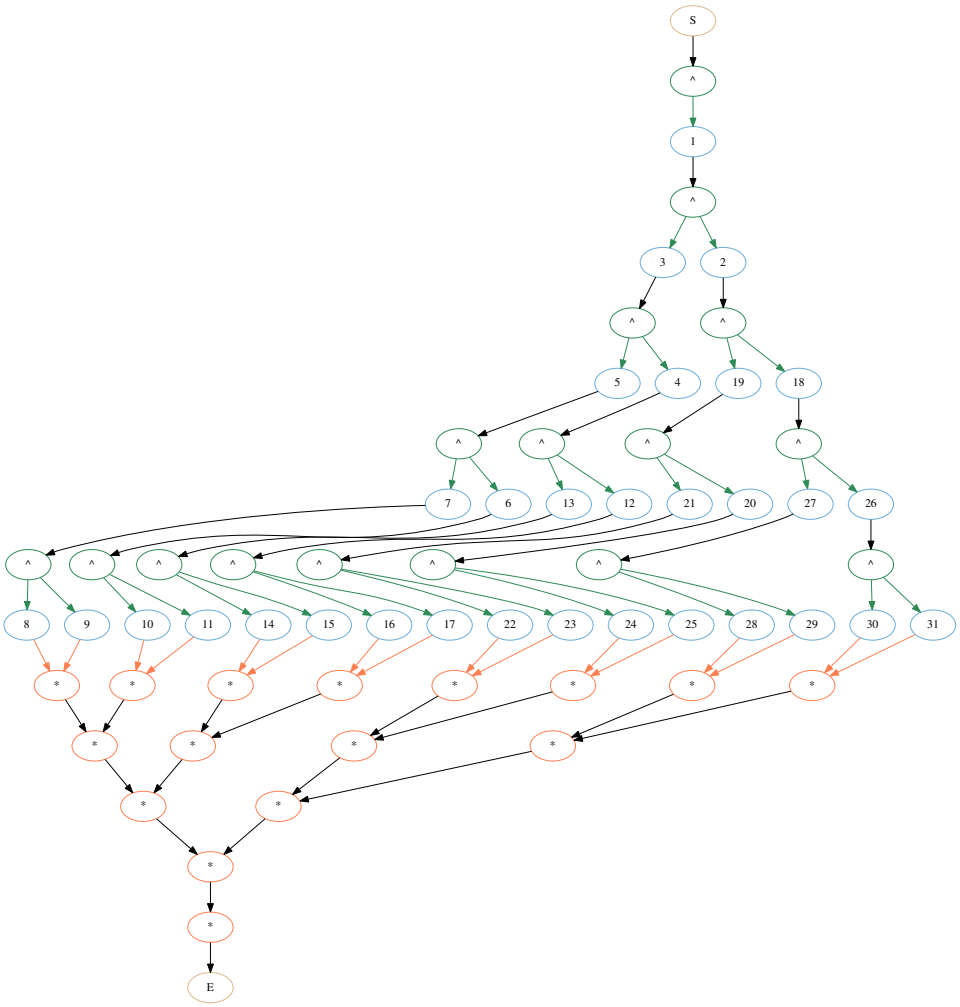

Figure 2: fib_test-annotated_task_graph.dot visualized using Dot