How can you make the prototypes to work?

Sreeram Sadasivam
M.Sc Distributed Software Systems
TU Darmstadt, Germany
sreeram.sadasivam@stud.tu-darmstadt.de

March 20, 2018

This document covers the details about how you can build the prototypes used for the IRS scheduler realized in kernel space. This document also covers the details about how you can get the evaluations done.

Prerequisties

The prototypes are only compatible with any 64 bit Linux operating system with a kernel version greater than 3.0 and which support loadable modules. For running the IRS framework, you would require LLVM-CLANG 3.9, GCC/G++ 4.9, BOOST 1.6.2, latest version of cmake and latest version of graphviz. You would additionally need the latest version of python and spdlog for generating debug logs(This prerequiste is only applicable if you are intending to run the evaluations).

Where can you find the Prototypes?

You can find all the prototypes inside the directory **Prototype_Impl** which deals with the implementations of various prototypes. Inside this folder, you would find two main sub-folders dealing with two approaches discussed in the thesis work related to these prototypes. These two approaches are realized in the folders: **no_check_userspace** and **check_userspace**.

Directory trees represented below shows us how the source files for the various Prototypes are mapped to the directory structure.

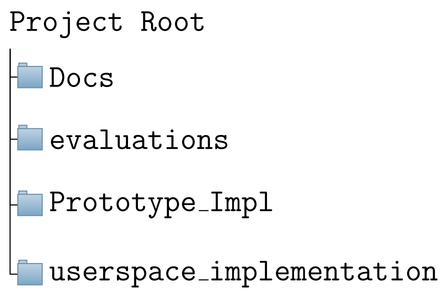

Figure 1: Directory tree from project root

Loading and Unloading the Prototype

Before we load the prototype to kernel space, we need to know the number of threads used in the multithreaded program. Currently, all the prototypes have the *THREAD_COUNT* parameter as statically defined. You have to modify the macro entry *THREAD_COUNT* inside the headerfile *common.h* with the number of threads you have in your multithreaded program. Once that step is complete, you can load the kernel module by running the makefile located in the root directory of the prototype with target load. The command *make load* will compile and load the prototype to kernel space with the number of threads you have indicated in the *common.h* headerfile. The command *make unload* will remove the module from the kernel space and clean up the module versions from your machine.

Loading the Prototype

- Move into the directory location **Prototype_Impl** where all the prototypes are mapped as shown in the directory tree 2.
- Move to the Prototype you need to load, complying the directory tree shown in figure 2.
- Inside every Prototype folder you would find the directory structure similar to the one shown in figure 3.
- Running the command *make load* will compile and load the prototype to kernel space with the number of threads you have indicated in the *common.h* headerfile.

Unloading the Prototype

- Move into the directory location **Prototype_Impl** where all the prototypes are mapped as shown in the directory tree 2.
- Move to the Prototype you need to load, complying the directory tree shown in figure 2.
- Inside every Prototype folder you would find the directory structure similar to the one shown in figure 3.
- Running the command *make unload* will remove the module from the kernel space and clean up the module versions from your machine.

Invoking Prototype related functions from a Multithreaded Program

We have five main functions which are defined in the headerfile *user_space.h*. These functions are $BeforeMA()$, $AfterMA()$, *initialize_trace*(), *reset_clock*() and *thread_reg*(). Currently, these prototypes are not integrated with LLVM therefore, methods such as $BeforeMA()$ and $AfterMA()$ needs to be invoked at the point of every shared memory event of the threads manually. But this functionality could be modified once integrated with LLVM. These two function expect the thread id which is value in the range of 1 to N. *initialize_trace*() is the first method which needs to be invoked from the multithreaded program. The method needs to be called before the creation of the threads and it expects a vector clock representation similar to the one described in the thesis, but without newline and white space characters. *thread_reg*() needs to be called first within the thread function for registering the thread and it expects a thread id similar to $BeforeMA()$. *reset_clock*() is the final method which needs to be used with the multithreaded program. This method needs to be used once all the threads have completed their execution. The program needs to be run in superuser mode because the IOCTL device created when the module was loaded requires root permission for access. The multithreaded program should only be compiled and executed until the all kernel modules of the prototype are loaded successfully.

Running Evaluations

Building the user space IRS solution

- Currently, the choice between *IRS_Opt* and *IRS_Sh*(two IRS user space solutions discussed in the thesis work) is not programmatically controlled. You would have go into the **userspace_implementation** folder. Move inside this location **MemoryChecksrcScheduler** where you find the file **Runtime.cpp**.

- Modify the line containing the following code:

  $scheduler.reset(newSharedScheduler(threadCount, traces));$

  to

  $scheduler.reset(newOptimizingScheduler(threadCount, *permissionManager));$

  if you need to change the IRS user space scheduler to use behave as *IRS_Opt* instead of *IRS_Sh*.

- To enforce this scheduler, you need to build the IRS framework.

- Run *mkdir build* inside the **MemoryCheck** folder.

- Move inside the build directory and run the command *cmake..*. Note: You need to have the latest version of cmake installed to run the IRS framework.

- Once this command is run successfully, run the command *make*. This command would build the IRS framework which would also include the benchmarks and tests added as a part of the framework. Therefore, this command would take some time to complete.

Running the benchmark

You can the run the existing benchmarks by moving into the folder location **experiments** under the **userspace_implementation** folder. You can find the binary executables with and without IRS of each benchmark inside this folder. These binary executables have a soft symbolic link inside the **experiments** folder, ensure that original binaries are generated at its source location of the link.

For running the benchmark inside the Prototypes, you would need to first ensure the Prototype has been loaded into the kernel space with the necessary steps adhered in the previous sections. Once the prototype module is loaded into the kernel, you can run the command *sudo make test_ < benchmark >*. The value of benchmark would be: last_zero, indexer_t15, dining_phil, fibonacci. Or, you run any script located inside the Prototype directory with the command *sh < benchmark_scr > .sh*. The value of benchmark_scr would be: last_zero, indexer_t15, dining_phil_prob, fibonacci. This script would automatically load the prototype into the kernel with the necessary configuration for the benchmark and compile and run the benchmark 1000 times. Once the executions are complete of the benchmark, you would get different files with .dat extensions. These files contain the execution time taken for each benchmark under a specific execution traces. If a benchmark is executed with 3 traces, there would be three dat files containing the results of execution time taken when run with these 3 traces.

Testing and Evaluation of Benchmark across IRS solutions

You can run an automated Makefile script located inside the **Prototype_Impl** folder. You can run the benchmark with the following command:

make test_and_eval_ < benchmark >.

The value of benchmark includes: indexer_t15, last_zero, dining_phil,fibonacci. Running the above command would internally trigger a script for running the provided benchmark across all the prototypes and also run inside the configured IRS user space scheduler. The results are generated and printed on to the terminal screen with evaluation results based on the metric - execution time overhead. The benchmark is executed 1000 times across each IRS solution.

Evaluation results used for the thesis

You can find the results of various benchmarks under the folder evaluations.

Note

The script used to run the IRS user space solution would only run the benchmark and not build the IRS framework. Therefore, you have to manually build the framework yourself with the necessary scheduler option. If you change the scheduler from *IRS_Opt* to *IRS_Sh*, you would have to rebuild the entire IRS framework which is described in the previous sections.

```
Prototype_Impl
├─ no_check_userspace
│  ├─ Prototype_1
│  ├─ Prototype_2
│  ├─ Prototype_3
│  └─ Prototype_4
├─ check_userspace
│  ├─ Prototype_5
│  └─ Prototype_6
├─ include
├─ Traces
├─ Makefile
├─ eval_dining_phil_prob.py
├─ eval_fibonacci.py
├─ eval_indexer.py
├─ eval_indexer_t15.py
├─ eval_last_zero.py
├─ test_dining_phil_prob_userspace.sh
├─ test_fibonacci_userspace.sh
├─ test_indexer_t15_userspace.sh
└─ test_last_zero_userspace.sh
```

Figure 2: Directory tree of Prototype_Impl

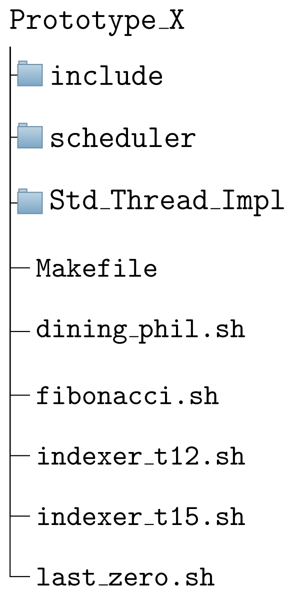

Figure 3: Directory tree of any Prototype