

Scaff User Manual

Embedded Systems Software Lab
Southern Illinois University, Carbondale

January 2021

Contents

1	About Scaff	3
1.1	Executor	3
1.1.1	Initialization	4
1.1.2	Execution Control	5
1.2	Scheduler	5
2	Installation	7
2.1	Install libraries	7
2.1.1	Known Shared Library Issue	7
2.2	Set Path	8
2.3	Installation Script	8
2.4	Architecture-Specific Settings	8
3	Files & Directory Structure	9
3.1	Directories	9
3.2	Scripts	9
3.3	Other Files	10
3.4	Changing Default Paths & Names	10
4	Compilation	10
4.1	Compiling Scaff	10
4.2	Compiling Benchmarks	10
4.2.1	Polybench Benchmark Suite	10
4.2.2	Stream Benchmark Suite	11
4.3	Setting Performance Counters	11
4.4	Architecture Specific Settings	11
4.5	Python Packages	12
5	Profiling Benchmarks	13
5.1	Storing Benchmark Statistics	13
5.2	Profiling Script	13
6	Running Experiments	14
6.1	Configuration File	14
6.2	Schedulers	15
6.3	Run Scaff	15
6.4	Results	16
7	Experiment Results	16
7.1	Result Directory Structure	17
7.2	Metadata	17
7.3	Raw Results	18
7.4	Summarized Results	18
7.5	Parsing Old Results	18
7.6	Searching Experiment Results	18

8	Using Cache Allocation Technology (CAT)	18
8.1	Determine CAT Support	18
8.2	Integrating CAT into Schedulers	19
9	Others	21
9.1	Isolating a socket	21
9.2	Using CPUID	21
9.3	Performance Monitoring	21
9.4	cpuset	22
9.5	freezer	22
10	Examples	23
10.1	Profiling Experiment	23
10.2	Normal Experiments	24
10.2.1	Static Scheduler	24
10.2.2	Dynamic Scheduler	26
11	Notes	27
12	Known Issues	27
12.1	Libcpuset/Libbitmask Error	27
12.2	GCC Installation Error on Comet Lake/ Core i9:	27
12.3	Repositories/GCC	27
A	Scaff Architecture Detailed Diagram	28
B	Server #1 Specific Settings	29
C	Server #2 Specific Settings	29

1 About Scaff

Scaff is a runtime system that orchestrates the execution of multithreaded applications. It operates on user-level space on top of Linux-based operating systems. Its primary role is to provide a communication mechanism between the applications being executed and a scheduler implementation. It relies on two subsystems, the *executor* and the *scheduler*. Figure 1 depicts a simplified version of scaff’s architecture. A more detailed diagram is presented in Appendix A.

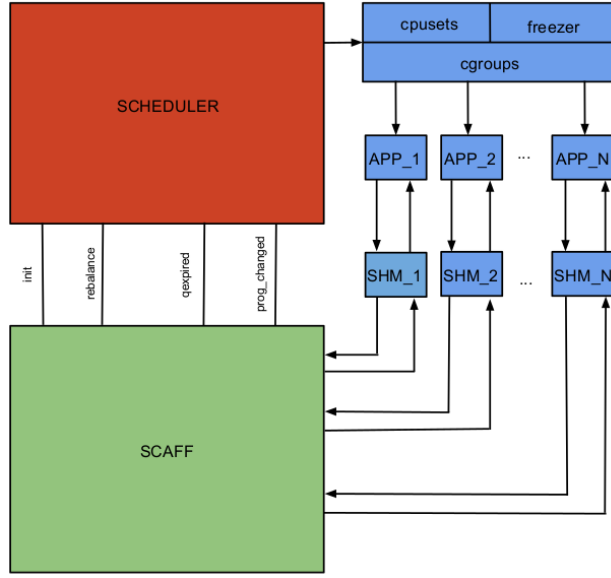


Figure 1: Scaff Architecture Overview

The *executor* is responsible for handling events regarding the execution, such as creation or termination of programs, and handling the communication between applications and the scheduler. It, also, provides an interface to assist the scheduler manage the workload.

Concerning the *scheduler*, it makes the decision on how the available resources should be distributed among the programs of the workload. This means that it implements a policy for choosing which applications should be assigned on which cores (space-sharing) on each time-quantum (time-sharing).

1.1 Executor

The executor takes four arguments. The first is a configuration file, which contains the desired applications associated with information about the resources they need (cores and memory nodes). The second one is the directory in which we print the results and debugging information. The third is a comma separated list of cores, on which we choose to assign the applications of our workload. On the last argument the name of the desired scheduler is given. Having acquired all this information, the executor is now ready to start the execution of the workload. Its work can be divided into two phases. The first is the *initialization* and the other (and final) is the *execution control*. Figure 2 shows an overview of the executor subsystem.

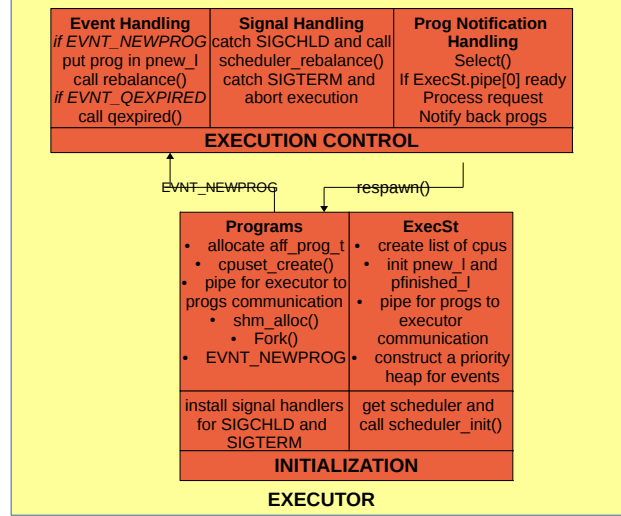


Figure 2: Executor Overview

1.1.1 Initialization

The global state of the executor is represented by the `ExecSt` struct. Its fields are initialized one by one. First we store the list of cpus we acquired from the input. Then we initialize two lists, one for the spawned programs (`pnew_l`) and one for the finished (`pfinished_l`). In order to achieve the applications→executor communication, we create a pipe, where the applications write on the write-end of the pipe and the executor reads from the read-end. As long as the main responsibility of the executor is to handle events, we construct a priority heap, where these events will be kept.

The executor parses the configuration file and begins execution of every program of the workload. For every program, it allocates and initializes a structure (`aff_prog_t`) that will be used to describe it throughout the execution. This structure contains a `cpuset_t` field that is used as a handler for the program's cpuset. The executor uses `cpuset_create()` to create a new cgroup to the `cpuset` virtual filesystem instance. The `aff_prog_t` also contains a pointer to the shared memory that the program will use to communicate with the executor. The executor initializes this portion of shared memory and sets some of the fields. These fields are the write-end of the executor's pipe that a program will use to notify about a new request, the number of cpus available, the cores allocated initially to the application and a pointer to the application's `aff_prog_t` structure that will be used as an identifier, when the program will make a request to the executor, so that the last will distinguish requests from different programs. Each program uses a pipe for the executor→programs communication. They block to the read-end of the pipe, waiting the executor to satisfy their requests. The executor writes some arbitrary values in the write-end pipe of the programs, after having processed their requests. In that way, we achieve a synchronous communication.

After the initialization of the new program's structure, the executor will `fork()` a new process. The new process uses the `exec1()` command to begin execution on a new shell. The executor waits for the program to freeze itself and then attaches to its `cpuset` all the cpus of the system. Finally an `EVNT_NEWPROG` event will be pushed on the heap and the program will remain `FROZEN` until the time that the event will be handled.

Every program is linked with a pre-load dynamic library. This library interacts with shared memory's fields and takes care of communication between the application and the executor

(`affhook_region_notify()`). In addition every application has its own ID field. We use the `/AFF_ID` name to distinguish cpusets, freezers and shared memory between different applications.

After parsing the configuration file, the executor initializes the scheduler and installs handlers for the `SIGCHLD` and `SIGTERM` signals. The first corresponds to normal termination of a program and will cause a subsequent call to the scheduler, while the other indicates unexpected termination and is handles as an error, causing execution abort.

1.1.2 Execution Control

Now that we have finished the initialization phase, the executor is responsible for handling iteratively two types of signals, two types of events and the programs' notifications.

As mentioned above, when a program terminates normally the `SIGCHLD` signal is caught by the executor, which in its turn calls the `rebalance()` function of the scheduler. There the scheduler handles the internal structures of the program and may need to make a new scheduling policy. The `SIGTERM` implies abnormal termination of the application and is considered as an erroneous behavior, causing the stop of the execution.

In addition, the executor pops events from the priority heap until it becomes empty. Every event is linked with a timestamp. The executor compares the current timestamp with that of the highest priority event and if it is time to process it, it pops it from the heap and handles it. If the event is of `EVNT_NEWPROG` type, the program is added to the `pnew_l` list of the `ExecSt` structure. On the other hand, when an `EVNT_QEXPIRED` event has arrived, this means that a time-quantum has expired and the executor calls the function `qexpired()` of the implemented scheduler. This function make decisions about which programs should be assigned on which core for the next time-quantum and push to the heap an `EVNT_QEXPIRED` event representing the next time-quantum. If there are no other events, and programs have been added to the `pnew_l` list, then the `rebalance()` function is called. Until the process of the next `EVNT_QEXPIRED` event, the executor waits for program notifications.

The executor uses the `select()` system call to check if there is a program notification. `Select()` receives a set of file descriptors (`fds`) as the first argument and a struct `timeval` as the second. `Select()` watches if some of the given `fds` becomes ready (characters available for reading or writing). If there is no change in the observed `fds`, `select` sleeps for the amount of time specified by the `timeval` and on wake up returns 0. Otherwise, it wakes up when a `fd` becomes ready and returns the `fd` value. In our case, the `fd` we want to watch until the next time-quantum, is the read-end of the executor pipe. So we give `ExecSt.pipe[0]` as the first argument and the timestamp of the next event as the second one. In that way we succeeded in polling for program notifications, until the next event in the heap must be handled. When a notification has arrived, the executor reads it from the read-end of its pipe.

1.2 Scheduler

The second subsystem of the Scaff tool is the scheduler and is responsible for the placement of the programs to the available cores. It determines on which time slice (time-sharing) and on which cores (space-sharing) each program should be executed. Figure 3 depicts an overview of the scheduler subsystem. It consists of the following functions:

1. **`init()`**: is called by the executor on the initialization phase. It allocates the structure, which represents the scheduler and is necessary for the management of programs. This struct is stored by the executor and used to subsequent calls. In addition the hardware performance counters are activated and set to zero.

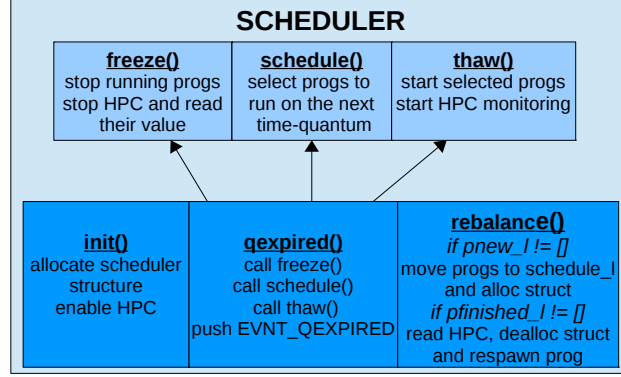


Figure 3: Scheduler Overview

2. **qexpired()**: is called by the executor at the expiration of the time-quantum (EVNT_QEXPIRED occurs). Subsequently, it calls the functions **freeze()**, **schedule()** and **thaw()**, making the decision for the next time-quantum. It also pushes the EVNT_QEXPIRED to the heap.
3. **freeze()**: first function called by the **qexpired()**. It is time for the running applications to leave the cores (freezer state set to FREEZE). The performance counters are stopped, read and their value is stored in the internal structure of each program.
4. **schedule()**: second function called by the **qexpired()**. There we choose which applications should run on the next time-quantum.
5. **thaw()**: third function called by the **qexpired()**. The selected applications start the execution at the available cores (freezer state set to THAW). The performance counters are started again.
6. **rebalance()**: called by the executor either when the *pnew_l* is not empty (EVNT_NEWPROG) or when a SIGCHLD arises. In the first case it moves the programs from the *pnew_l* list to the *app_l* and allocates for every program a structure that the scheduler uses to represent them. In the other case, the program is removed from the *pfinished_l* list and is respawned if this attribute is enabled.

2 Installation

This section will guide you through the process of setting up the Scaff tool from a cloned repository.

2.1 Install libraries

Use the following commands to install each of the prerequisite libraries in `HOME_DIR/rpm_build/`
For Ubuntu:

```
sudo apt-get install libnuma-dev libnuma1
sudo apt-get install alien
cd rpm_build/
sudo alien -i *.rpm
cd ../
```

NOTE: If a "cannot locate libcputset.so" error occurs when attempting to run Scaff, uninstall the packages listed above and then re-install them using the apt-get commands listed below.

```
sudo apt-get install libbitmask1
sudo apt-get install libbitmask-dev
sudo apt-get install libcputset1
sudo apt-get install libcputset-dev
```

For CentOS:

```
sudo yum install numactl numactl-devel numactl-libs numad
cd rpm_build/
sudo rpm -i *.rpm
cd ../
```

2.1.1 Known Shared Library Issue

When executing SCAFF, the following (or similar) errors may show up on some computers while compiling or attempting to execute Scaff:

```
-> error while loading shared libraries: libcputset.so.1: cannot open
shared object file: No such file or directory
->error while loading shared libraries: libbitmask.so.1: cannot open
shared object file: No such file or directory
->libbitmask.so => not found
```

To fix this error, try to use one of the following solutions, arranged in decreasing order of preference:

1. [Best method] Add the correct folder to `$PATH`.

```
export $PATH=$PATH:/lib/x86_64-linux-gnu/
```

2. [Not as good, but might work] Copy the installed libraries to the system's default folder:

```
cp /usr/lib64/libcputset* /lib/x86_64-linux-gnu/
cp /usr/lib64/libbitmask* /lib/x86_64-linux-gnu/
```


2.2 Set Path

In the main Scaff directory, open `home_dir.py` and set the variable `HOME_DIR` to the absolute path of the Scaff directory on your computer (add the `'/'` at the end of the path) . This only needs to be done once and will help with automation of experiment results' storage, indexing, and parsing.

2.3 Installation Script

Load msr module (i.e. `modprobe msr`) and mount the `cpuset/freezer` subsystems (one-time run after boot).

Jump to `HOME_DIR/scripts/` and open `prepare.sh`. Set the `'USER'` variable to your username.

```
./prepare.sh
```

2.4 Architecture-Specific Settings

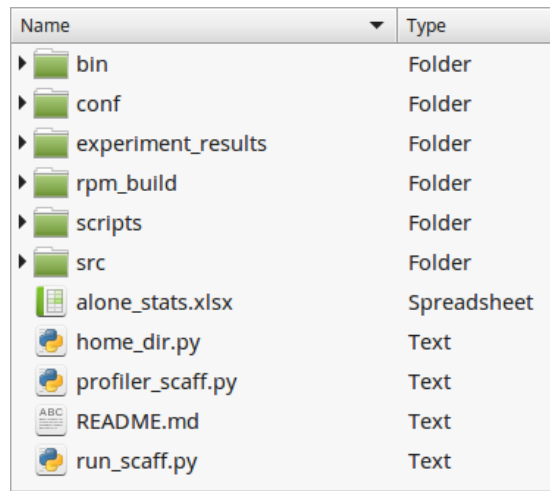
To complete the installation process, jump to `HOME_DIR/src/prfcnt/prfcnt_xeon_types.h` and set the hexadecimal values for your processor's specific Model Specific Registers (MSRs).

To find the exact MSR names and addresses for your CPU, refer to [Intel's Software Developer Manual](#), Volume 4, Chapter 2¹. The use of Intel's Memory Bandwidth Monitoring (MBM)/Memory Bandwidth Allocation (MBA) also relies upon correct MSR values. If the MSR addresses are mismatched, the results shown can be incorrect.

¹<https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>

3 Files & Directory Structure

Figure 6 shows the files and directories included in the default Scaff repository.



Name	Type
bin	Folder
conf	Folder
experiment_results	Folder
rpm_build	Folder
scripts	Folder
src	Folder
alone_stats.xlsx	Spreadsheet
home_dir.py	Text
profiler_scaff.py	Text
README.md	Text
run_scaff.py	Text

Figure 4: Home directory for Scaff repository

3.1 Directories

The Scaff home directory (hence referred to as `HOME_DIR`) consists of the following sub directories:

- `rpm_build`: Use this directory for installing prerequisite libraries
- `bin`: This is where benchmark executables are stored
- `conf`: This is where the ‘configuration file’ is stored
- `src`: Contains the main code for the executor and schedulers
- `experiment_results`: Each experiment will be saved in a new directory here
- `scripts`: Contains utilities for installation, cleanup, parsing, etc.
- `scripts/scheds`: Contains source code for all schedulers
- `scripts/prfcnt`: Performance counters can be configured from the files stored here

3.2 Scripts

The following scripts are used for running experiments.

- `home_dir.py`: This script stores the absolute path to Scaff’s home directory.
- `profiler_scaff.py`: Used for profiling individual benchmarks (section 5).
- `run_scaff.py`: Wrapper script for running experiments.

Furthermore, the following scripts are included for support and additional functionality:

- `scripts/cleanup.sh`: Destroy zombies, thaw freezer, clear cpuset. This is done automatically for each experiment but can also be called from here.
- `scripts/hyperthreading.sh`: Use this script to determine which cores are hyperthreaded.
- `scripts/parse_results.py`:
- `scripts/utilities.py`: Contains default names and helper functions.
- `scripts/search_experiments.py`: Use to search old results. See section 7.5 for usage details.

3.3 Other Files

The Scaff home directory also contains an Excel file named `alone_stats.xlsx`. This file contains execution statistics for individual benchmarks when running without co-runners for one complete period of execution. Section 5 describes the use of this file along with the process for automatically profiling applications.

3.4 Changing Default Paths & Names

By default, Scaff will use the directories listed in Section 3.1 to store the configuration files, results, benchmarks, etc. If required, these values can be reconfigured from `utilities.py`. Furthermore, the default naming scheme for experiment result directories can also be changed from this script.

4 Compilation

4.1 Compiling Scaff

Jump to `HOME_DIR/src/` directory and build the tool

```
cd src/
make
```

4.2 Compiling Benchmarks

This section describes the process of compiling benchmarks included with the master scaff repository.

4.2.1 Polybench Benchmark Suite

Jump to `HOME_DIR/bin/polybench-c-4.2.1-beta/` directory

Usage:

```
./compile_polybench benchmark_domain
```

Example:

```
./compile_polybench bench_1          # compile all benchmarks
./compile_polybench datamining_1     # compile only one domain
```

4.2.2 Stream Benchmark Suite

Jump to `HOME_DIR/bin/stream/` directory

Usage:

```
./compile_stream.py [-h] [--dataset DATASET] [--times TIMES] [--openmp OPENMP]
```

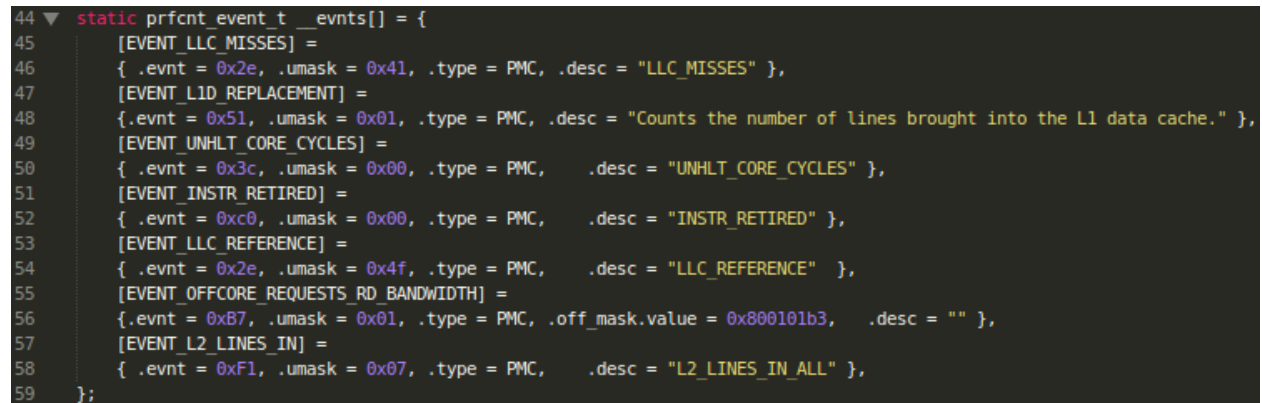
Example:

```
./compile_stream.py --dataset 128M --times 10 --openmp no
```

4.3 Setting Performance Counters

Performance counters can be configured in `HOME_DIR/src/prfcnt/prfctr_xeon.events.h`. There are two types of performance events - fixed and programmable.

In lines 44-61, replace the hexadecimal values with the ones for your specific architecture. Figure 5 shows the struct `prfcnt_event_t__events` which holds the information for PMC events. To find the correct values for your CPU, refer to [Intel's Software Developer Manual](#)² Volume 3, Sections 18.3 to 18.6. For more information about performance monitoring, refer to Section 9.3.



```
44 static prfcnt_event_t __events[] = {
45     [EVENT_LLC_MISSES] =
46     { .evnt = 0x2e, .umask = 0x41, .type = PMC, .desc = "LLC_MISSES" },
47     [EVENT_L1D_REPLACEMENT] =
48     { .evnt = 0x51, .umask = 0x01, .type = PMC, .desc = "Counts the number of lines brought into the L1 data cache." },
49     [EVENT_UNHLT_CORE_CYCLES] =
50     { .evnt = 0x3c, .umask = 0x00, .type = PMC, .desc = "UNHLT_CORE_CYCLES" },
51     [EVENT_INSTR_RETIRED] =
52     { .evnt = 0xc0, .umask = 0x00, .type = PMC, .desc = "INSTR_RETIRED" },
53     [EVENT_LLC_REFERENCE] =
54     { .evnt = 0x2e, .umask = 0x4f, .type = PMC, .desc = "LLC_REFERENCE" },
55     [EVENT_OFFCORE_REQUESTS_RD_BANDWIDTH] =
56     { .evnt = 0xb7, .umask = 0x01, .type = PMC, .off_mask.value = 0x800101b3, .desc = "" },
57     [EVENT_L2_LINES_IN] =
58     { .evnt = 0xf1, .umask = 0x07, .type = PMC, .desc = "L2_LINES_IN_ALL" },
59 };
```

Figure 5: Performance Monitoring Counter Events

4.4 Architecture Specific Settings

Different architectures will vary in how they read memory bandwidth. Run the following command to see which PCI addresses are in use:

```
cat /proc/bus/pci/devices
ls /proc/bus/pci/
```

Update the hexadecimal address of the memory controller in `HOME_DIR/src/prfcnt/prfctr_xeon.types.h`.

Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz [COMET LAKE]:

Socket0: 00

²<https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>

Socket1: 04

Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz [PUGET/Haswell]:

Socket0: ff

Socket1: 7f

In `HOME_DIR/src/scheds/common.h`, update the number of available CPUs.

```
#define MAX_CORES      5
```

4.5 Python Packages

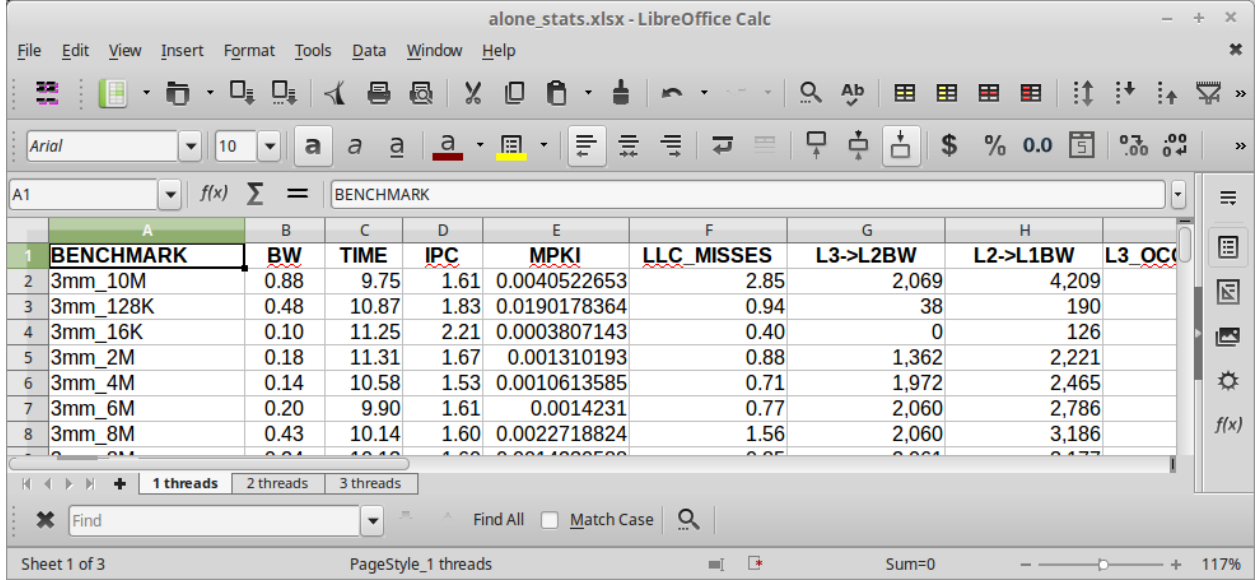
Install the following packages using pip.

1. numpy
2. pandas
3. xlrd
4. xlswriter

5 Profiling Benchmarks

5.1 Storing Benchmark Statistics

For many experiments, it is beneficial to compare the results of application execution to a ‘baseline’ case. In this manual, we consider the baseline to be an application’s execution statistics (such as IPC, exec time, etc.) obtained from running it for one full execution without any interference from co-running applications (i.e. alone execution on the socket/CPU).



	A	B	C	D	E	F	G	H	I
1	BENCHMARK	BW	TIME	IPC	MPKI	LLC_MISSES	L3->L2BW	L2->L1BW	L3_OCC
2	3mm_10M	0.88	9.75	1.61	0.0040522653	2.85	2,069	4,209	
3	3mm_128K	0.48	10.87	1.83	0.0190178364	0.94	38	190	
4	3mm_16K	0.10	11.25	2.21	0.0003807143	0.40	0	126	
5	3mm_2M	0.18	11.31	1.67	0.001310193	0.88	1,362	2,221	
6	3mm_4M	0.14	10.58	1.53	0.0010613585	0.71	1,972	2,465	
7	3mm_6M	0.20	9.90	1.61	0.0014231	0.77	2,060	2,786	
8	3mm_8M	0.43	10.14	1.60	0.0022718824	1.56	2,060	3,186	

Figure 6: alone_stats.xlsx

Scaff can automatically normalize certain results with respect to the ‘baseline’ if the statistics for a benchmark are already present in the Excel file `alone_stats.xlsx` included in `HOME_DIR`. This file is used to keep track of all alone execution statistics and is organized into different sheets, one per number of threads used in the profiled benchmark. This is necessary since a given benchmark performs differently when executing with different numbers of threads. When parsing results, the number of threads is automatically taken into account.

5.2 Profiling Script

For quickly obtaining execution statistics for a particular benchmark when executing alone, the script `profiler_scaff.py` can be used. This script executes the given benchmark until completion while recording the counts of PMC events. It requires the following inputs in lines 39-42:

- **benchmark** This is the application to be executed until completion. Make sure it is located in the `bin/` folder.
- **num_threads** The number of threads to compile the benchmark with (OpenMP supporting benchmarks only)
- **cores** The set of cores we want to dedicate to this workload. Note that the number of cores specified must be equal to or greater than the number of threads. Unused cores will not be

assigned any applications for the duration of the profiling experiment.

- **comments** The comments will be written to file in the experiment folder.

6 Running Experiments

The following sections describe the process of running Scaff with user-defined schedulers to collect performance data.

6.1 Configuration File

To specify the workloads that SCAFF must run, they must be listed in a tab-separated text file. The directory `<SCAFF_dir>/conf` contains two examples of configuration files. One for memory-intensive (`'mb_prevalent'`) and one for cache-intensive workload (`'cs_prevalent'`). Each application is specified using a new line. Lines beginning with `'#'` are for comments and are ignored when parsing the config file. Each line in the configuration file corresponds to one application, and consists of the following tab-separated fields:

1. **Cores needed:** The number of cores required to execute this application
2. **Gang id:** Used for static scheduling when applications need to be executed in gangs/groups/batches.
3. **Core id:** The core(s) to be assigned to the application. For multi-threaded applications requiring more than one core, this field contains the index of the starting core. For dynamic schedulers, this field can be set to a placeholder value such as 0 for all lines in the config file.
4. **IPC/BW alone:** This field is used by some schedulers to determine run-time groupings. It can be set to the alone IPC or memory bandwidth for the corresponding application. Set to 0 if the scheduler does not require this information.
5. **Start time:** The start time of an application. Normally this is 0 but can vary with scheduler implementations.
6. **Path:** Relative path to the executable, along with any command line arguments for the application.
7. **Priority:** For use with priority-aware schedulers. Set to a placeholder value if not required.
8. **Min ratio:** For use with priority aware schedulers. Set to a placeholder value if not required.
9. **Max ratio:** For use with priority aware schedulers. Set to a placeholder value if not required.
10. **Name:** The name of the application, as it will appear in the results. Furthermore, if a matching name is found in the file `HOME_DIR/alone_stats.xlsx` then the summarized results will automatically also contain normalized results.

Listing 1 depicts an example configuration file for the static scheduling policy included in the Scaff repository `cgang`. For this example, we execute two groups (`'gangs'`) of four applications each.

Listing 1: Example Config File

```

1      0      0      0      0      ../ bin/ft.S.x      0      0      0      ft.S
1      0      1      0      0      ../ bin/bt.W.x      0      0      0      bt.W
1      0      2      0      0      ../ bin/atax_6M 20      0      0      0      atax_6M
1      0      3      0      0      ../ bin/gemm_4M 50      0      0      0      gemm_4M
# This is a comment line
1      1      0      0      0      ../ bin/3mm_32K 50      0      0      0      3mm_32K
1      1      1      0      0      ../ bin/2mm_32K 70      0      0      0      2mm_32K
1      1      2      0      0      ../ bin/gemm_32K 8      0      0      0      gemm_32K
1      1      3      0      0      ../ bin/mg.W.x      0      0      0      mg.W

```

Important Notes:

1. Configuration files MUST end in a newline character.
2. Not every scheduler utilizes all fields of the configuration file.

6.2 Schedulers

The default Scaff repository includes two schedulers: linux and cgang. However, any number of user-defined schedulers can be used with Scaff as long as they use the interface described in Section 1.2 to communicate with the executor subsystem. We have successfully implemented and tested the following schedulers using Scaff:

- **Linux:** Completely fair scheduler
- **Cgang:** Simple static scheduler
- **Perf:** [Link to paper.](#)³
- **Perf&Fair:** [Link to paper.](#)⁴
- **BAOS:** [Link to paper.](#)⁵
- **Proposed Dynamic:** Resource aware scheduler

6.3 Run Scaff

In the main directory, open the ‘run_scaff.py’ script. In line #2 of the code, set the variable HOME_DIR to the path to the Scaff main directory. Don’t forget to add the ‘/’ at the end of the path.

In the main function, the script takes 4 inputs:

- **config_file:** The configuration file where the workload is defined (check HOME_DIR/conf/ for examples)
- **cores:** The set of cores we want to dedicate to this workload (lscpu for more info)
- **scheduler:** The scheduling policy utilized to control the execution (options: linux_cfs, cgang)

³<https://ieeexplore.ieee.org/document/7161508>

⁴<https://ieeexplore.ieee.org/document/7676358>

⁵<https://www.computer.org/csdl/journal/tc/2016/02/07100868/13rRUx0gez8>

- **comments:** The comments will be written to file in the experiment folder

Additional options in `HOME_DIR/src/scheds/cgang.c` and `HOME_DIR/src/scheds/linux.c`:

- **time_window:** If this is set then execution will stop when `tics=run_tics`
- **run_tics:** If `time_window` is `True`, Scaff will stop execution when `tics=run_tics`

Example input:

```
config_file=HOME_DIR+'conf/cs_prevalent'
cores='0-5'
scheduler='linux_cfs'    # Options linux_cfs, cgang
exp_comments=''          # Any additional comments
```

6.4 Results

A new folder is created in `HOME_DIR/experiment_results` for each experiment. In each folder, the 'about.txt' file contains a description of the experimental setup, date/time, and additional notes (if any). The results from each experiment are automatically parsed and summarized in an excel file (.xlsx) in the experiment's main folder. This file contains three sheets:

- **Counters:** Per-application summary of the execution statistics.
- **Finisheds:** Per-application summary of run times, waiting times, finish times, etc.
- **Overhead:** The scheduling overhead incurred (in micro seconds) at each time quantum

If needed, the raw results can be found in the `HOME_DIR/raw_results` sub directory of each experiment folder. It contains the following files:

- **counters-out:** Contains performance counter and energy values for each time quantum
- **finisheds-out:** Every time an application finishes, its execution time and related stats are logged here.
- **energy-out:** Contains the energy consumed by package and DRAM during each time quantum.
- **overhead.csv:** Contains the scheduling overhead incurred between each successive time quanta (micro sec)
- **scaff-out:** Contains a log of executor functions such as memory allocation, etc.
- **sched-out:** Contains a log of the scheduling decisions undertaken by the scheduler at the expiration of each time quantum.

7 Experiment Results

This section presents an in-depth description of how Scaff manages experiment results.

7.1 Result Directory Structure

By default, experiment results are stored in `HOME_DIR/experiment_results` unless specified otherwise in `HOME_DIR/scripts/utilities.py`. A new directory is created to store the results and metadata for each new experiment. Directory names have the default format `scaff_exp-<exp_num>`. This can be also be changed from the `utilities_and_paths.py` script. Scaff utilizes a text file stored in the experiment results directory to keep track of experiment numbers. This file is called `experiment_number.txt`. If this file is missing, Scaff will automatically generate a new file restarting the numbering from 0.

7.2 Metadata

Scaff stores information for each experiment inside the experiment's directory. The configuration file is automatically copied to the result folder in case it needs to be referenced again. In addition, the file `about.txt` is created for each new experiment. This file contains the following information:

```
1
2  # ----- About SCAFF Exp#3669 ----- #
3
4  config file: /hdd1/shivam/SCAFF_Shivam/conf/g28
5  cores: 0-5
6  scheduler: cgang
7  comments: This experiment is for group #28
8
9  Description of counters:
10 IRETD:      Instructions Retired in Giga
11 CYCLES:     CPU Cycles in GHz
12 LLC_MISSES: LLC Misses in Kilo
13 MPKI:      Misses per Kilo Instructions
14 MEM_BW(C): Per Core Memory Bandwidth in MB/s
15 MEM_RD BW(S): Socket Read Memory Bandwidth in MB/s
16 MEM_WR BW(S): Socket Write Memory Bandwidth in MB/s
17 L3->L2 BW:  L3->L2 Bandwidth in MB/s
18 L2->L1 BW:  L2->L1 Bandwidth in MB/s
19 IPC:       Instructions per Cycles
20 L3_OCC:     L3 Occupancy in KB
21 TEMP(C):   Core Temperature in Celcius degrees (Tjmax = 85, TCCoffset = 0)
22 TEMP(P):   Package Temperature in Celcius degrees
23
24 start time: Oct-19-2020 17:01:49
25 end time: Oct-19-2020 17:02:06
26 total time taken: 16.58 seconds
```

Figure 7: Sample about.txt file

7.3 Raw Results

7.4 Summarized Results

7.5 Parsing Old Results

The script `scripts/parse_results.py` can be used to re-parse any old experiment result. The experiment number can be specified by command line argument.

Usage:

```
./parse_results <exp number>
```

7.6 Searching Experiment Results

The default Scaff repository contains a helper script `scripts/search_experiments.py`. The purpose of this script is to crawl through the auto-generated `about.txt` files, containing the description of the experimental setup, date/time, and additional notes(if any), and return the experiment numbers with a matching field. Experiments can be searched using the following fields:

1. exp type
2. config file
3. cores
4. scheduler
5. comments

8 Using Cache Allocation Technology (CAT)

Intel's Cache Allocation Technology allows for partitioning of the LLC's available cache-ways among the available cores.

8.1 Determine CAT Support

In order to find if the architecture supports CAT, follow the instructions in section 4.1 to compile the Scaff tool and then jump to `HOME_DIR/src/`. Run the executable `intel_technologies`. The resulting output will show if CAT is supported on the system.

Example Output:

```
#####  
#                               CPU CHARACTERISTICS                               #  
#####  
# Vendor:                       GenuineIntel  
# Model name:                   Intel(R) Xeon(R) Gold 6130 CPU  
                                @ 2.10GHz  
# cpuid:                        50654  
# Family:                       6  
# Microarchitecture codename:   Skylake-SP  
# MSR support:                  YES  
# Hyperthreading support:       YES
```

```

# Perf Version: 4
# General Purpose Counters Number: 4
# Gpcounters Width: 48
# Fixed Counters Number: 4
# Fcounters width: 32
# Cache Monitoring Technology: YES
# Memory Bandwidth Monitoring (Total): YES
# Memory Bandwidth Monitoring (Local): YES
# L3 Cache Allocation Technology: YES
# L2 Cache Allocation Technology: NO
# Memory Bandwidth Allocation Technology: YES
# Code and Data Prioritization Technology: YES
# Turbo mode: OFF
#####

```

8.2 Integrating CAT into Schedulers

By default, the Scaff tool provides a mechanism to schedulers for managing shared memory and other execution characteristics, while also recording the counts of selected PMC events. The use of CAT can be integrated into any user-defined scheduler that conforms to the communication scheme presented in Section 1.2. The following steps describe the process of using CAT with Scaff schedulers:

1. Find the # of cache ways supported by the architecture. For example, on a Xeon Gold 6130s (SkyLake) the LLC is an 11-way set-associative cache. Furthermore, the minimum # of cache ways that can be assigned to a core in this architecture is 2.
2. Find the available # of Classes-of-Service (CoS) for the architecture. Each CoS represents a particular cache configuration by means of the corresponding bitmask. The easiest way to find # of CoS is to use the `intel-cmt-cat` tool available [here](https://github.com/intel/intel-cmt-cat)⁶.

The sample output presented below shows that our system supports 16 Classes-of-Service per node. Each cache-way is represented by 1 bit in the bitmask. For example, the bitmask `0x7ff` (`0b01111111111`) represents all 11 cache-ways being available, the bitmask `0x600` (`0b01100000000`) represents 2 cache-ways being available, and the bitmask `0x1ff` (`0b00111111111`) represents 9 cache-ways available.

Example Output:

```

L3CA COS definitions for Socket 0:
L3CA COS0  => MASK 0x600
L3CA COS1  => MASK 0x1ff
L3CA COS2  => MASK 0x7ff
L3CA COS3  => MASK 0x7ff
L3CA COS4  => MASK 0x7ff
L3CA COS5  => MASK 0x7ff
L3CA COS6  => MASK 0x7ff
L3CA COS7  => MASK 0x7ff

```

⁶<https://github.com/intel/intel-cmt-cat>

```

L3CA COS8  => MASK 0x7ff
L3CA COS9  => MASK 0x7ff
L3CA COS10 => MASK 0x7ff
L3CA COS11 => MASK 0x7ff
L3CA COS12 => MASK 0x7ff
L3CA COS13 => MASK 0x7ff
L3CA COS14 => MASK 0x7ff
L3CA COS15 => MASK 0x7ff

```

3. Allocate the required cache-configuration to an available COS using the relevant bitmask.
4. Associate the COS to the required core(s). All applications running on the associated core will have access to the assigned cache-ways of the corresponding COS.

The sample association output presented below shows that cores 0 to 8 have access to the cache-ways represented by COS1, i.e. 9 available ways. The other cores are associated with COS0, which includes 2 cache-ways. In this particular configuration, all processes executing on cores 0,2,4,6,8 have exclusive access to 9 cache-ways and the rest of the cores have access to 2 cache-ways. Note that the bitmasks do not overlap, meaning that the available cache-ways to each COS are exclusive.

Example Output:

```

Core information for socket 0:
  Core 0 => COS1
  Core 2 => COS1
  Core 4 => COS1
  Core 6 => COS1
  Core 8 => COS1
  Core 10 => COS0
  Core 12 => COS0
  Core 14 => COS0
  Core 16 => COS0
  .
  .
  .
  Core 62 => COS0

```

5. To perform these steps at runtime, the scheduler must integrate the association and allocation functions in its `schedule()` function (see Section 1.2 for details). During execution, control is handed over from the executor to the scheduler (at the end of each time quantum) in order to decide which processes need to be executed next. The scheduler already possesses the updated counts of all recorded PMC events when execution of `schedule()` function begins. Depending on the schedulers functioning, performance requirements of applications, and resource availability of the system, the scheduler can invoke the COS association and core-to-COS allocation functions to assign the needed cache-configurations to cores. Furthermore, the COS associations and core allocations can also be reset during this stage if required. Note

that re-configuring cache-allocations results in significantly increased overhead compared to simple resource-aware decision making.

9 Others

9.1 Isolating a socket

When running an experiment, it is beneficial to isolate the cores (or socket) to prevent interference from other applications and kernel tasks running in the background. We use the `cset` command to move all the applications from one socket to another. That way, the socket we want to utilize will be “almost” free of applications and we can run our workload.

Note that the specific cores in the `cpuset` will depend on the architecture of the platform. For example, on an Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz (Haswell microarchitecture) with 24 cores and 2 NUMA nodes, we have cpus 0-5 (and their hyperthreaded cpus 12-17) on one NUMA node, and cpus 6-11 (and 18-23 hyperthreaded) on the other node. We isolate the socket in the following way:

```
# create "system" cpuset with [6-11,18-23] cpus
cset set -c 6-11,18-23 -s system

# list the cpusets
cset set -l

# move processes from "root" to "system" cpuset
cset proc -m -f root -t system

# move kernel threads from "root" to "system" cpuset
cset proc -k -f root -t system

# list the processes located in "root" cpuset
cset proc -l -s root

# move bash instance (from which scaff is executed) to the "root" cpuset
cset proc -m -p $$ -t root
```

9.2 Using CPUID

The `CPUID` instruction, short for CPU-Identification, is used to enumerate the various capabilities and architectural features for Intel processors. feature information to the EAX, EBX, ECX, and EDX registers. For more information, see [Intel’s Software Developer Manual](https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html) ⁷, Volume 2A, Chapter 3.2.

9.3 Performance Monitoring

Intel processors contain two classes of performance monitoring events: architectural and non-architectural. Availability of architectural performance monitoring capabilities is enumerated using

⁷<https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4.html>

the CPUID.0AH.

The architectural events are consistent across platforms

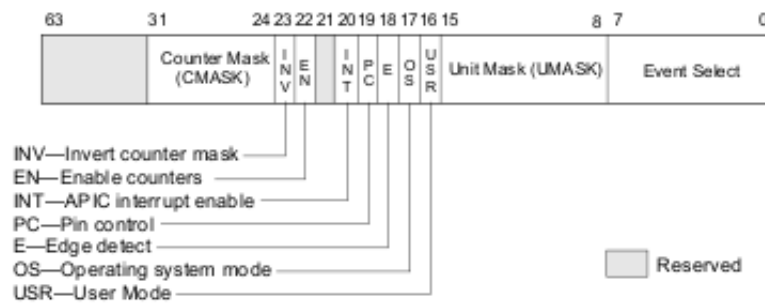


Figure 8: Layout of IA32_PERFEVTSELx MSRs

9.4 cpuset

The cpuset pseudo-filesystem can be used to isolate cores from outside interference. See [documentation](https://man7.org/linux/man-pages/man7/cpuset.7.html)⁸ here.

9.5 freezer

Scaff utilizes the freezer subsystem to support scheduling and performance monitoring. For more information, see [documentation](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-freezer)⁹ here.

⁸<https://man7.org/linux/man-pages/man7/cpuset.7.html>

⁹https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/sec-freezer

10 Examples

This section demonstrates the process of using Scaff to profile benchmarks, run scheduling experiments, and parse and search experiment results. The following examples require Scaff to be set up according to the instructions presented in Section 2.

10.1 Profiling Experiment

The Scaff tool includes a dedicated script `profiler_scaff.py` for quick and easy profiling of benchmarks. The specified benchmark will be executed for one full period with the specified number of threads. The cores should be equal to or greater than the number of threads of the selected benchmark. If there are excess cores, the profiler will keep them unassigned for the duration of the experiment to ensure no interference from other kernel/user tasks. Figure 9 shows the options used to profile the benchmark `correlation` (from Polybench benchmark suite¹⁰) with a dataset of 8MB and problem size of 15. The `benchmark` field contains the execution command, along with any required command line inputs. Figure 10 shows the summarized result for this experiment, located in a new folder created in the `experiment_results` directory.

```

36 ▼ if __name__ == '__main__':
37
38     # ----- User input ----- #
39     benchmark='correlation_8M 15'
40     num_threads=1
41     cores='0,1,2,3,4,5,12,13,14,15,16,17'
42     exp_comments='Testing polybench benchmarks' # Any additional comments
43     # -----

```

Figure 9: Example profiling experiment

INDEX	Benchmark	ALONE IPC	DRAMENRGY	IPC	IRETD	L2->L1BW	L3->L2BW	L3->L1BW
1	correlation_8M 15	N/A	0.9605340825	2.2001406907	1044058826.60825	17298.6769083505	1139.5052236392	9287
2								
3								
4								
5								
6								
7								

Figure 10: Profiling Experiment Result

¹⁰<https://web.cse.ohio-state.edu/pouchet.2/software/polybench/>

10.2 Normal Experiments

The examples presented in this section are for a Xeon processor with 12 physical cores partitioned into two 6-core sockets (each connected to a NUMA node). We utilize the cores of only one socket to ensure isolation from any unrelated kernel/user tasks. See Section 9.1 for instructions on how to isolate a socket.

We execute the following group of 12 applications (Table 1), taken from Polybench (4.2.1b)¹¹ and Stream¹² benchmark suites using the static scheduler **cgang** and the dynamic scheduler **linux.cfs** (both are included in the default Scaff repository). Four of the benchmarks are multi-threaded (**lu**, **3mm**, **stream 32M**, and **stream 64M**) while the rest are single-threaded, resulting in a total of 18 threads for the entire workload.

Index	Application	#Threads Needed	Index	Application	#Threads Needed
1	symm	1	7	2mm	1
2	lu	2	8	heat-3d	1
3	atax	1	9	durbin	1
4	gemm	1	10	stream 32M	2
5	doitgen	1	11	3mm	2
6	mvt	1	12	stream 64M	4

Table 1: Benchmarks to be scheduled

10.2.1 Static Scheduler

Static schedulers map applications/processes to cores before starting execution. The following steps describe how the scheduler **cgang** (included in the default Scaff repository) can be used to execute the workload listed in Table 1.

Since our system has only 6 physical cores, we split the benchmarks into three distinct groups (‘gangs’) that will execute interchangeably in a time multiplexed manner. For this example, we want assign benchmarks 1 to 5 to gang #1, benchmarks 6 to 10 to gang #2, and benchmarks 11 & 12 to gang #3. Complete the following steps to run this experiment:

1. Compile each benchmark and place the generated executables in an appropriate folder. For this example, we utilize the `<HOME_DIR/bin>` folder. Instructions for compiling Polybench & Stream benchmarks can be found in Section 4.2.
2. Create a configuration file in `<HOME_DIR/conf>` using the format presented in Listing 2 with each benchmark being specified in a new line (see Section 6.1 for more details). Figure 11 shows the completed configuration file for this workload. Since **cgang** does not require columns 7, 8, and 9, we set each value in these columns to a placeholder (‘0’ in this case).

Listing 2: Column Format

```
#cores | gang id | core id | alone | start_t | path | priority | min_r | max_r | name
```

¹¹<https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/>

¹²<https://www.cs.virginia.edu/stream/ref.html>

```

1  1  0  0  0  0  ../symm_2M 100  0  0  0  symm_2M
2  2  0  1  0  0  ../lu_8M 50  0  0  0  lu_8M
3  1  0  3  0  0  ../2mm_16K 5000 0  0  0  2mm_16K
4  1  0  4  0  0  ../heat-3d_16K 0  0  0  heat-3d_16K
5  1  0  5  0  0  ../atax_2M 10  0  0  0  atax_2M
6  # Comment line
7  1  1  0  10  0  ../durbin_4M 20  0  0  0  durbin_4M
8  1  1  1  10  0  ../gemm_128K 100  0  0  0  gemm_128K
9  2  1  2  10  0  ../stream_32M_OMP 0  0  0  stream_32M_OMP
10 1  1  4  10  0  ../doitgen_16K 200 0  0  0  doitgen_16K
11 1  1  5  10  0  ../mvt_32K 75  0  0  0  mvt_32K
12 # Comment line
13 4  2  0  20  0  ../stream_64M_OMP 0  0  0  stream_64M_OMP
14 2  2  4  20  0  ../3mm_2M 100  0  0  0  3mm_2M
15

```

Line 15, Column 1 Tab Size: 4 Plain Text

Figure 11: Configuration file for example #1

3. For each benchmark, we specify the executable's path relative to the `<HOME_DIR/src>` folder. In addition, provide any additional command line arguments such as problem size, input file, etc., required for each benchmark.
4. For multi-threaded benchmarks such as `lu`, `3mm`, and `stream`, specify the starting core id in the 'core' column. `cgang` will automatically check the number of threads and assign the rest of the cores incrementally. For example, for the double-threaded benchmark `lu`, we specify '1' as the starting core. Since it requires a total of two threads, the scheduler will assign them both consecutively to cores 1 and 2.
5. Make sure the configuration file ends in a new line.
6. Once the configuration file has been made, open `HOME_DIR/run_scaff.py` and fill in the details shown in Figure 12.

```

106 if __name__ == '__main__':
107
108     # ----- User input ----- #
109     scheduler='cgang'
110     config_file=HOME_DIR+'conf/example1_config'
111     cores='0,1,2,3,4,5'
112     exp_comments='This experiment is for example 1' # Any additional comments
113     # ----- #

```

Line 114, Column 1 Tab Size: 4 Python

Figure 12: Settings for example #1

7. Execute the `HOME_DIR/run_scaff.py` script. Once the experiment is complete, the results can be found in a newly made directory in `HOME_DIR/experiment_results/`.

10.2.2 Dynamic Scheduler

This example demonstrates how the workload presented in Table 1 be scheduled using the dynamic scheduler `linux_cfs` (included in the default Scaff repository). Dynamic schedulers determine the application-to-core placement at runtime.

1. Compile each benchmark and place the generated executables in an appropriate folder. For this example, we utilize the `<HOME_DIR/bin>` folder. Instructions for compiling Polybench & Stream benchmarks can be found in Section 4.2.
2. Create a configuration file in `<HOME_DIR/conf>` using the format presented in Listing 3 with each benchmark being specified in a new line (see Section 6.1 for more details). Figure 13 shows the completed configuration file for this workload. `linux_cfs` and other dynamic schedulers only require the number of threads, path, and benchmark name.

Listing 3: Column Format

```
#cores | gang id | core id | alone | start_t | path | priority | min_r | max_r | name
```

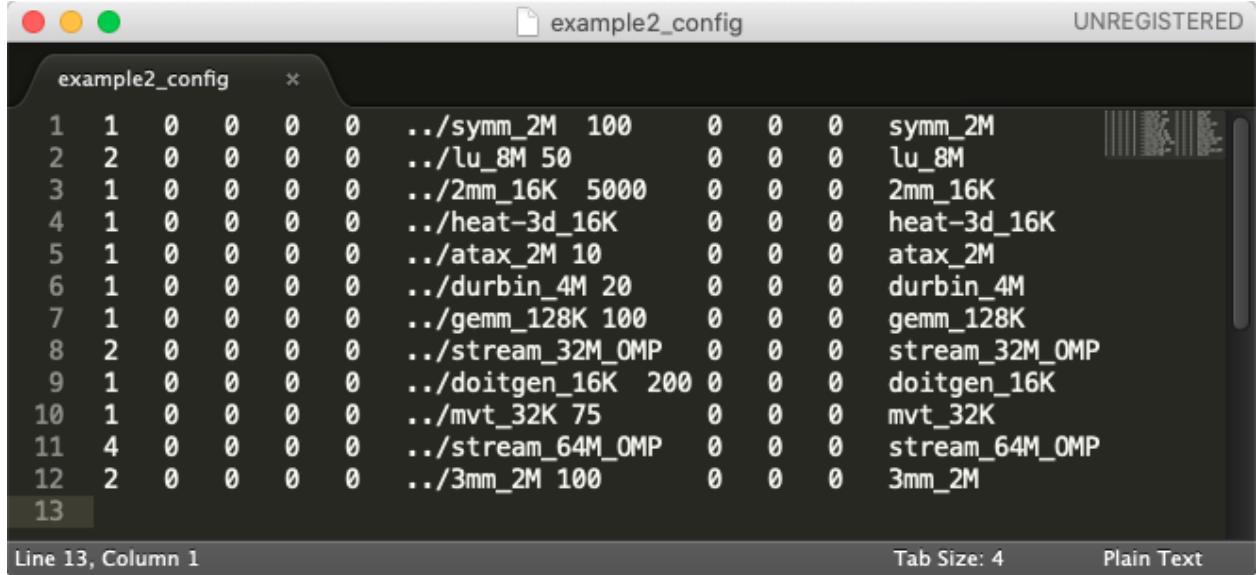


Figure 13: Configuration file for example #2

3. For each benchmark, we specify the executable's path relative to the `<HOME_DIR/src>` folder. In addition, provide any additional command line arguments such as problem size, input file, etc., required for each benchmark.
4. Make sure the configuration file ends in a new line.
5. Once the configuration file has been made, open `HOME_DIR/run_scaff.py` and fill in the details shown in Figure 14.

```

106 if __name__ == '__main__':
107
108     # ----- User input ----- #
109     scheduler='linux_cfs'
110     config_file=HOME_DIR+'conf/example2_config'
111     cores='0,1,2,3,4,5'
112     exp_comments='This experiment is for example 2' # Any additional comments
113     # ----- #

```

Line 114, Column 1

Tab Size: 4 Python

Figure 14: Settings for example #2

6. Execute the `HOME_DIR/run_scaff.py` script. Once the experiment is complete, the results can be found in a newly made directory in `HOME_DIR/experiment_results/`.

11 Notes

This tool has been tested on a Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz server (24 cores in total, Haswell microarchitecture) that supports Cache Monitoring Technology (CMT). The operating system run on this server is CentOS Linux release 7.4.1708. One numa node was utilized with hyper-threading disabled (i.e. cores 0-5). In core architectures performance monitoring capabilities are more limited. Run `intel_technologies` in `src/` directory to check what is supported in your system.

This tool records the aggregate socket bandwidth along with the per-core bandwidth. The aggregate bandwidth is extracted via reading the IMC registers from the PCI. In order to run in a different system one should check the `PCI.bus.device.function` and change the `SOCKET0_BUS`, `MC0.CHANNEL0.DEV.ADDR`, `MC0.CHANNEL0.FUNC.ADDR` values in `"prfctrn_xeon_types.h"`. Otherwise aggregate socket bandwidth should be disabled by commenting the `InitPower()`, `startPower()`, `startMCCounters()`, `stopMCCounters()` functions in the scheduler file.

12 Known Issues

12.1 Libcpuset/Libbitmask Error

12.2 GCC Installation Error on Comet Lake/ Core i9:

E: Failed to fetch http://archive.ubuntu.com/ubuntu/pool/main/l/linux/linux-libc-dev_5.4.0-42.46_amd64.deb
 404 Not Found [IP: 91.189.88.152 80] E: Unable to fetch some archives, maybe run `apt-get update` or try with `-fix-missing`?

I do not think this is the correct solution. We fixed it like that:
`sudo apt-get -o Acquire::Check-Valid-Until=false -o Acquire::Check-Date=false update`

<https://junise.wordpress.com/2016/07/26/ubuntu-restore-default-repository/>

12.3 Repositories/GCC

<https://junise.wordpress.com/2016/07/26/ubuntu-restore-default-repository/>
`apt-get install build-essential`

A Scaff Architecture Detailed Diagram

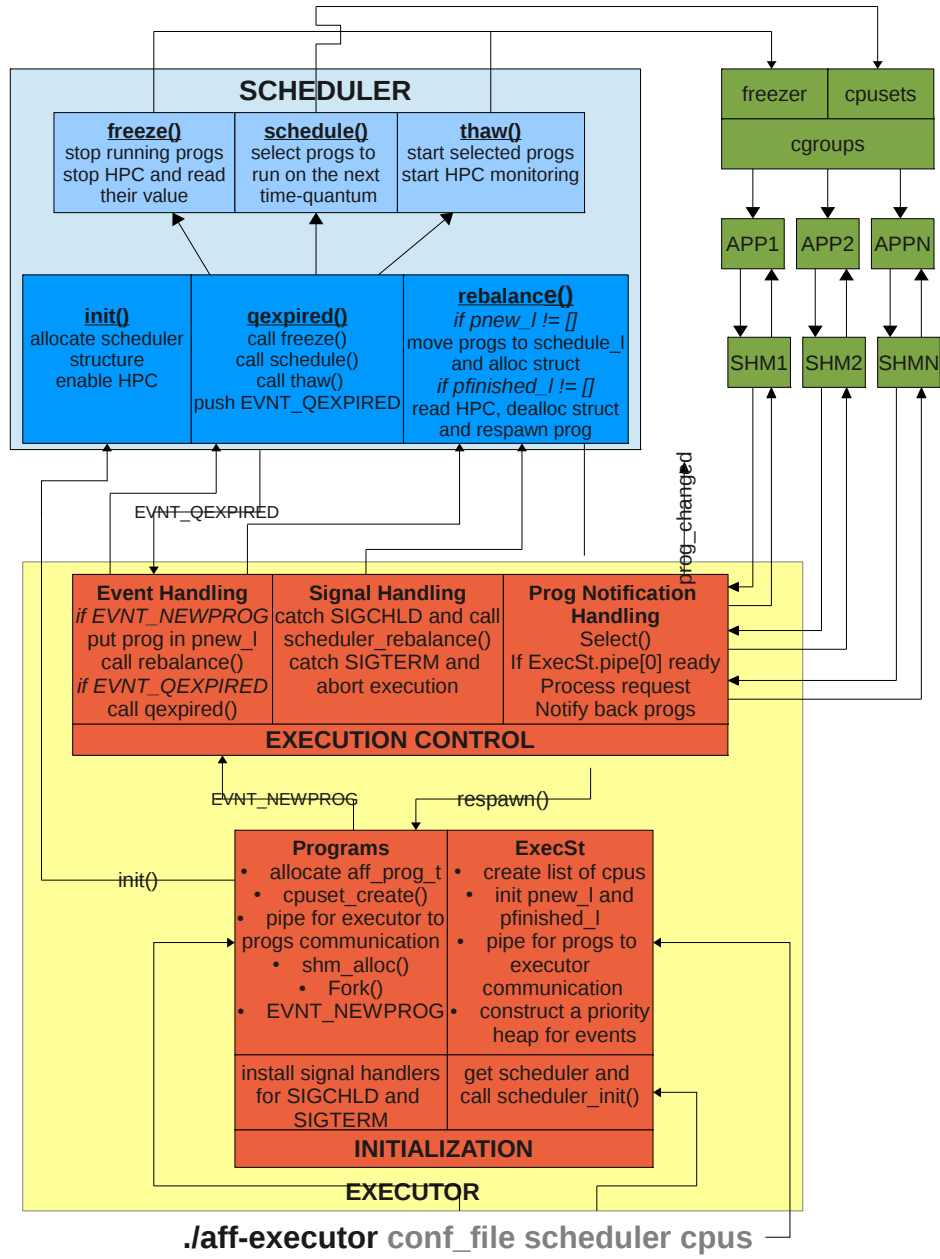


Figure 15: Detailed Scaff Overview

B Server #1 Specific Settings

Settings for puget server with IP address 131.230.192.5

Open `HOME_DIR/src/prfcnt/prfcnt.h` and change references to:

`HOME_DIR/src/prfcnt/prfcnt_sandy_types.h` and

`HOME_DIR/src/prfcnt/prfcnt_sandy_events.h`

C Server #2 Specific Settings

Settings for server with IP address 131.230.193.222

Open `HOME_DIR/src/prfcnt/prfcnt.h` and change references to:

`HOME_DIR/src/prfcnt/prfcnt_xeon_types.h` and

`HOME_DIR/src/prfcnt/prfcnt_xeon_events.h`