Simultaneous Multithreading and Hard Real Time: Can it be Safe? Artifact Evaluation

This document describes the experiments conducted in the paper "Simultaneous Multithreading and Hard Real Time: Can it be Safe?" by Sims Osborne and James Anderson, to be presented at ECRTS '20. Table 1 (q_c(1000) values), Fig. 4 (histogram of $M_{i:j}$ values) and all graphs in the paper can be duplicated by following these instructions, along with Tables 2, 3, and 4 and additional graphs included in the online appendix available at http://jamesanderson.web.unc.edu/papers/. The duplications for Tables 1-4 and Fig. 4 may not be exact, particularly if a different hardware architecture is used, but the same general trends should be evident. Testing on a virtual machine may cause more significant variations, as virtual machine treatment of hardware threads varies widely.

Benchmarks were tested on an Intel Xeon Silver 4110 2.1 GHz (Skylake) CPU running Ubuntu 16.04.6. Data analysis was performed with Python 3. Fig. 4 was created using Microsoft Excel, but could easily be created by any software package with statistical capabilities, including Python. We expect benchmarks will run on any Hyperthreading-enabled Intel platform (this includes most Xeon and Core processors) running Linux, but results will vary with a different platform. Running on a virtual machine will not produce the same results.

Schedulability tests were performed using Gurobi (www.gurobi.com), a commercial mathematical optimization solver. A free, full-featured academic license is available. The front-end for the Gurobi solver was programmed in Python 3, using the gurobipy module. Instructions for installing gurobipy are available as part of the overall Gurobi installation instructions provided on the Gurobi website.

Running Benchmark Tests (Folder benchmarkExecution)

All files discussed in this section are contained in the folder

Enabling and Disabling Hyperthreading

The instructions in this section previously appeared in the ECRTS 2019 paper "Simultaneous Multithreadng Applied to Real Time" (S. Osborne, J. Bakita, and J. Anderson).

Linux identifies processors with the following information that can be viewed by running cat /proc/cpuinfo.

Linux maintains a unique processor number for all hardware threads in the system. With Hyperthreading enabled, sibling processors can be identified as the processors which share a common physical id (i.e. socket number) and core id, but have distinct processor numbers. With hyperthreading disabled, there will be only one processor id corresponding to each unique physical id/ core id combination. Typically, sibling threads will have processor numbers that differ by the total number of physical cores (i.e. on a 16-core system, processors 0 and 16 are siblings), but this relationship should be verified prior to running benchmark experiments.

Hyperthreading can be enabled and disabled under advanced boot options. Once Hyperthreading is enabled in the boot menu, it can be turned off without rebooting by running the provided script benchmarkExecution/threadsOff.sh and enabled again by running the script benchmarkExecution/threadsOn.sh. The scripts assume a system of 16 physical cores with sibling threads having processor IDs separated by 16; if this is not the case on the target platform, the scripts will need to be edited.

The Benchmark Programs

Table 1 and Fig. 4 in the paper report results based on our modified versions of the TACLeBench sequential benchmarks.

The folder benchmarkExecution/sourceCode contains the benchmarks. It consists of 23 folders plus the files extra.h, which contains macros that are added into the benchmarks to assist with testing and a file listing all benchmarks to be executed, tacleNames.txt

Within each benchmark folder, the main file has the name [folderName].c. We have modified the benchmarks by adding input arguments, placing the main function in a loop, and outputting timing results of each loop to a text file. In addition, all caches are cleared between loops, ensuring each loop runs on a cold cache. The majority of the modifications to the benchmark programs are contained in the macro file extra.h.

Each modified benchmark accepts the following parameters. While the benchmark programs can be run as freestanding programs for for the baseline case where SMT is not enabled, the intention is that programs should be run via the scripts baseline.sh and allPairs.sh (elaborated on below). Running benchmark programs with SMT enabled directly, rather than through allPairs.sh, may produce undefined behavior.

- **maxJobs** determines how many jobs the program executes, assuming output=1 (see below).
- **thisProgram, thisCore, otherCore**, and **otherProgram** are used to make the output more readable. They are not used in any decision making; all decisions based on those variables are handled by the bash scripts which run the relevant tests.
- **runID** is used to define the name of the output file and also prints as part of the output.
- **output:** if set to 1, the program will loop maxJobs times, killing the cache at the beginning of each loop and recording the time taken for each loop after the first (in nanoseconds). The first loop is not timed, but is used to make sure all relevant data is in memory; once brought into memory all data is kept there by the mlockall() system call. At the end of each loop, outside the timer, the program saves the recorded time. At the end of maxLoops, the program will kill all programs whose PIDs were passed as input parameters and print output to the file runID.txt. If output is set to 0, the program will not time itself, will not kill the cache, and will not give output. It is expected that if output=0, maxJobs will be an arbitrarily large value and the program will be terminated by receiving a kill signal from another program. Output=0 is not used in this work; the variable is a legacy of our previous work submitted to ECRTS ‘19 (S. Osborne, J. Bakita, and J. H. Anderson. Simultaneous multithreading applied to real time. In ECRTS 2019.)

- A string listing PIDs that should be killed upon program termination. This parameter has no effect and can safely be omitted if output=0 OR if any co-scheduled program is set to terminate at the same time. As with output=0, this variable is not used in this work, but is a legacy of the same earlier work.

Scripts governing benchmark execution

baseline.sh and allPairs.sh are designed to execute all benchmark programs without SMT, to obtain baseline measurements, and all pairs of benchmarks using SMT.

baseline.sh takes the arguments **firstCore, secondCore, maxJobs,** and **runID.** firstCore determines the core to be used, maxJobs determines how many jobs are to be run. runID and secondCore are passed to the benchmark program and included in the output file. We recommend using "none" or something similar for secondCore. It will execute all benchmarks listed in tacleNames.txt, each on firstCore, maxJobs times. All output from benchmarks executed by baseline.sh will appear in a single file, runID.txt.

allPairs.sh accepts the same parameters as baseline.sh. Whereas baseline.sh executes each program maxJobs times, allPairs.sh executes every possible pair---including programs paired with a second copy of themselves---maxJobs times. For correct execution, firstCore and secondCore must give the processor numbers of two hardware threads that share a single physical core. baseline.sh produces two output files, runID-A.text and runID-B.txt. Two obtain the joint costs for each pair, runID-A and runID-B are processed by a separate program discussed below.

Compilation

The following bash script will compile all benchmarks and place them in the desired destination folder. It is expected that tacleNames.txt (included in interference-benchmark) is in the location from which the command is run and that extra.h is in the folder interference-benchmark. An example script is given in benchmarkExecution/compileAll.sh (note compileAll.sh gives a path specific to our setup).

```
while read t; do
        gcc /path/benchmarkExecution/sourceCode/$t/*.c –o /destFolder/$t
done <tacleNames.txt
```

Note tacleNames.txt does not include all 23 benchmark programs; four of the programs--anagram, audiobeam, g723_enc, and huff_dec--were excluded from compilation and execution, since they could not be adapted to execute in a loop as we required. A copy of tacleNames.txt, along with baseline.sh and allPairs.sh, must be placed in destFolder prior to executing the compiled benchmarks.

Benchmark execution

Before the benchmarks can be executed, a small kernel module must be added to allow for cache invalidation.

From within the folder wbinvd, run the command “make”, then “sudo insmod wbinvd.ko.”

If users do not wish to add a kernel module, the benchmarks should be executable without it by deleting the statement FLUSH_CACHES from the definition of START_LOOP in line 71 of extra.h and recompiling the benchmarks as described above. However, doing this will cause benchmark results to be significantly different than what we report.

All measurements were conducted on a system that was idle apart from automatic system processes. Results may vary in the presence of other work.

To measure executing times without SMT, disable Hyperthreading and execute the command

sudo chrt –f 1 ./baseline.sh *firstCore secondCore maxJobs runID*.

(note that baseline.sh will have to first be given execution permission by running chmod 755 baseline.sh) in the folder to which the benchmarks were compiled (destFolder above). The destination folder needs to contain a copy of the folder tacleNames.txt. Running the script will execute every benchmark maxJobs times and output the time for each run to the file r*unID*.txt. Our experiments used coreID=0 and maxJobs=100,000. We recommend first performing a trial run with maxJobs=10.

maxJobs=100,000 may require a prohibitively long time to run; users may wish to limit themselves to smaller values of maxJobs and/or edit tacleNames.txt so that they do not execute all the tacleBench jobs we considered. If users choose to execute only a subset of the jobs we list in tacleNames.txt, additional edits will be needed to the programs described under benchmark analysis below.

In addition to the cost of actually executing each benchmark, clearing the cache between each benchmark loop may require several milliseconds per job. Evaluators should keep this in mind before running larger test sets, as for most of our benchmarks, clearing the cache took significantly longer than the actual timed execution.

To measure execution times with SMT, enable Hyperthreading and then execute the command

sudo chrt –f 1 ./allPairs.sh *firstCore secondCore maxJobs runID*

where firstCore and secondCore give the coreIDs of two threads that share a physical core. We used firstCore=0, secondCore=16, and baseJobs=100,000. As with baseline.sh, we recommend a trial run with maxJobs=10. Executing allPairs.sh with maxJobs=100,000 could potentially take several days.

Analyzing Benchmark results (folder benchmarkAnalysis)

In the previous section, we described how to execute all benchmarks to produce timing data. Here, we describe how to analyze that data. All files and programs discussed in this section are contained in the program benchmarkAnalysis.

Our own benchmark output is contained in the following files. Please not these files are not included in the github repository due to file size restrictions. They can be obtained from http://jamesanderson.web.unc.edu/papers/ and are included in the official artifact available through http://www.dagstuhl.de/darts.

- Jan11_100k.txt (baseline results)
- Dec14-100k-A.txt
- Dec14-100k-B.txt
- Dec19-100k-A.txt
- Dec19-100k-B.txt

The Dec14 and 19 files collectively contain a full set of results for allPairs. They are split into two parts due to a power interruption that occurred while running allPairs.sh the first time. We include two python programs to summarize this data; one accounts for our data being split across more files than intended, and the other summarizes data in the format we expected, i.e. allPairs.sh should produce two output files, runID-A.txt and runID-B.txt.

evaluateQProvidedData.py evaluates our raw benchmark data. It produces two output files, which we include in benchmarkAnalysis: baseline.csv gives results for baseline execution times, and allPairs.csv gives results for SMT execution times. Table 1 shows excerpts of allPairs.csv. Tables 2 and 3 (in the online appendix) show excerpts from baseline.csv. Fig. 4; the histogram of M_{i:j} values; is also based on allPairs.csv. It can be constructed by opening allPairs.csv in a spreadsheet Program (we used Microsoft Excel), filtering out terms for which costRatio>10 holds, and creating a histogram from the remaining M_{i:j} values (denoted by the column heading “score” in allPairs.csv).

evaluateQCustomData.py is provided for the evaluation of benchmark data users may produce on their own. Input file names need to be provided in lines 46, 48, and 49 of the code. It differs from evaluateQProvidedData.py in that it assumes only two files for the allPairs data, runID-A.txt and runID-B.txt. All other aspects of the program are identical. To run the program, first enter appropriate file names at the top of the program. If you have run data using only some of the provided benchmark programs, you will need to edit the list “benchmarkNames” near the top of the program to include only the programs for which you have data. If you ran benchmarks using fewer than 1500 trials, you will need to edit the values for medSampleLength and shortSampleLength (also near the top of the program) so that both values are less than the total number of benchmarks you ran. Output will be placed by default in the files customBaseline.csv and customAllPairs.csv.

In both the .csv files we provide and any files produced from custom data, the following patterns should hold: First, the majority of q_c(shortSample) values will be less than the bound given by q_b. Recall that q_b is a function of both the total number of jobs executed and the sample size; its is given by Exp. (7) in the paper. Second, the majority of M_{i:j} (i.e. score) values will fall between 0 and 1, particularly when considering only task pairs for which C_i/C_j (costRatio in allPairs.csv) is no more than 10. These two trends support our assertions that 1) task costs can be measured as reliably with SMT as without and 2) employing SMT generally reduces total needed execution times. Note that these two trends, and the assertions they support, remain in place after the corrections discussed above.

Synthetic Task Creation and Testing (folder scheduler)

The actual scheduler for CERT-MT consists of four python programs. To run, users must first install Gurobi Optimizer---a free academic license is available---and gurobipy, its python front end.

RunAllTests.py accepts the input parameters M, MIN_SAMPLES, SYM_DISTRIB, SYM_PARAM1, SYM_PARAM2, SYM_PARAM3, PERIOD_COUNT, and FILE_OUT.

· M---core count

· MIN_SAMPLES---approximate number of samples to generate for each total utilization range (we used 50; evaluators may wish to use a smaller value for testing).

· SYM_DISTRIB---2 for split-uniform, or 4 for split-normal.

· SYM_PARAM1---lower distribution bound for uniform or split-uniform distributions, mean for normal or split-normal distributions.

· SYM_PARAM2--- upper distribution bound for uniform or split-uniform distributions, standard deviation for normal or split-normal distributions.

· SYM_PARAM3---split for either split distribution, ignored otherwise.

· PERIOD_COUNT---how many choices of period to allow a task system. The results in the paper were produced with this value set to 4.

· FILE_OUT---the output file.

Invoking this program will produce a single output file giving schedulability results corresponding to the input parameters with each of the low, medium, high, and wide per-task utilization ranges.

As discussed in the paper, SYM_DISTRIB and SYM_PARAM1, 2, and 3 are used to synthetically model how SMT may affect task timings. The values we give for these parameters are based on our results contained in allPairs.csv, but evaluators should feel free to experiment with other values for these parameters.

The parameters we used are listed in the paper. RUN WITH CAUTION; in some cases, even on a machine with 24 cores, running a single instance of RunAllTests.py required over 2 days. (the name runAllTests is a misnomer; producing all graphs in the paper requires running RunAllTests for every combination of

SYM_DISTRIB, SYM_PARAM1, and SYM_PARAM3 listed in the paper, for both 4 and 8 cores). We recommend an initial run with 4 cores and a value of 10 for minSamples.

RunAllTests outputs a single file for each instance of the program; each file contains data for the light, medium, heavy, and wide per-task utilization ranges.

The file **jobScript.txt** contains commands that will run all tests on a research cluster using the SLURM job submission system. Note that the core and memory requirements listed in our commands are based on the user limits allowed on the system longleaf.unc.edu and on the assumption that all commands in the file will be run in parallel. For this reason, along with the large amount of time and resources required, we DO NOT RECOMMEND running jobs exactly as listed in jobScript.txt. We provide to show all the parameter combinations for which we ran RunAllTests.py and as a starting point for duplicating schedulability tests on using a slurm-based system. RunAllTests.py can also be run directly; a SLURM-based system is useful but not required.

CERTMT_runScenario.py creates task systems and runs schedulability tests for a single scenario. It is called by RunAllTests.py, or can be run directly for a single scheduling scenario. For testing purposes, we recommend the latter. The parameters in the included main method are simplified compared to what was in the paper; they should produce a program that completes relatively quickly, but is complex enough to give some insight into the program's operation. None of our reported results involved running this program directly, but we did so during writing and debugging.

CERTMT_TaskSystem.py creates the task systems that are then tested for schedulability. It is called by CERTMT_runScenario. It calls each of two scheduling methods to test a given task system for schedulability. If lines 134-138 and 143-148 are uncommented, it will print the solution found for each task system, allowing users verify correctness.

CERTMT_sched.py tests a given task system for schedulability according to CERT-MT using Gurobi Optimizer as a back-end. It includes the ability to print a found solution.

baseline_sched.py tests a given task system for schedulability according to a traditional cyclic-executive scheduler that does not allow SMT. It is not discussed in the paper, as a cyclic-executive scheduler that allows unlimited preemptions is very similar in practice to an ideal scheduler.

Creating Schedulability Graphs

After running schedulability tests as described above, running the program genGraphsECRTS.py will generate graphs illustrating each schedulability test. The assumption is that all files created by RunAllTests.py are in a subfolder titled "results" belonging to the same folder that holds genGraphsECRTS.py. Graphs will appear as pdf files in the folder results/graphs.

We include our own data in the folder “results.” Running genGraphsECRTS.py without changing the contents of results will reproduce all graphs contained in the paper and online appendix. The python file assumes the exact file names we have used; running it with different file names will require some edits to the code beginning at line 150. Note that file names are also used to generate graph headers.

In order for the headings on individual graphs to be correct, files containing schedulability results must be named using the same format that is used in jobScripts.txt; each output file begins with MCores_DParam1-Param2-Split where M gives the core count and D is replaced with U (for uniform distributions) or N (for Normal distributions).