Proactive and Adaptive Energy-Aware Programming with Mixed Typechecking

We are submitting all collected data, the source code for our compiler and runtime, and our benchmark suite for Intel, Raspberry Pi, and Android.

1. Prerequisites

We are providing two files: (1) a tar of the compiler and runtime uploaded directly, and (2) a link to the benchmark suite including data due to its large file size (5 GB).

We use `TOP` to refer to the untared benchmarks suite, and `ENT_HOME` to refer to the untared compiler. We make use of `ruby` scripts extensively and require both `ruby` and `gem` to be installed.

Our compiler and runtime assume a system that is compatible with `ACPI`, `MSR`, and `RAPL`. Any system with a Sandy Bridge chipset or later should be compatible. Please install ACPI (`sudo apt-get install acpi`). We provide instructions for managing `MSR` under the Intel section.

Our experiments were run across three platforms: Intel, Raspberry Pi, and Android. In each case the hardware requirements and the software configurations are described in the individual sections. Should any questions arise when constructing the hardware and software environment please send us a message for clarification.

Fresh Install Setup The following is a set of instructions to help get ENT installed on a fresh ubuntu system. We require Java 1.8 and ruby 2.1 or greater with `JAVA_HOME` set. We have provided some steps to help get these dependencies installed with `apt-get`; depending on your system, you may not need to add the repositories. We assume the following commands are executed from `ENT_HOME`.

- `sudo apt-add-repository ppa:brightbox/ruby-ng`
- `sudo add-apt-repository ppa:openjdk-r/ppa`
- `sudo apt-get update`
- `sudo apt-get install openjdk-8-jdk`
- At this point, please set `JAVA_HOME` to `/usr/lib/jvm/java-8-openjdk-amd64`.
- `sudo apt-get install ruby2.4*`
  % note that this is installing ruby and ruby-dev
- `sudo apt-get install gem`
- `sudo apt-get install zlib1g*`
- `sudo gem install ptools`
- `sudo gem install nokogiri`
- `sudo gem install terminal-table`
- `sudo apt-get install ant`
- `sudo apt-get install git`
- `sudo apt-get install subversion`
- `cd /vendor/jrapl-port ; make clean; make ; cd ../..`
- `ant clean; ant jar`
- At this point, please export `ENT_HOME` and add `ENT_HOME/bin` to your path.
- `cd /tests ; ./test.bash`

2. Data

Directory Structure Our applications have multiple benchmarks which correspond to the benchmarks described in our paper: (1) battery-casing, (2) battery-exception, and (3) temperature-casing. We provide a rough overview of the structure of our benchmark suite in Figure 1. We will continue to refer back to this table to help avoid confusion.

Data Format & Files We generate energy files for battery-casing and battery-exception benchmarks. If you examine the sample file listed in Figure 1 you will see several lines of the format `ERun 0: 10 20 30 40`. This corresponds to one run of the benchmark's energy readings, which translates to uncore GPU (J), CPU (J), package (total) (J) and runtime in seconds. Note that this is for our ivy bridge architecture and may differ for your system. Please see the jRAPL repository for further details.

We help link the cases for battery-casing and battery-exception runs discussed in our paper using Figure 2 and Figure 3. We use short labels for the file names and sometimes for brevity when displaying data. Typically, ld, md, and sd for large, medium, and small data or workload modes, and hc, mc, lc or full_throttle, managed, and energy_saver boot modes. For battery-exception ent runs and ent silent runs, we add the 'p', and 'u' respectively.

We generate timestamp data for temperature-casing runs which consist of temperature recordings per task. We help link the cases for temperature-casing runs using Figure 4.

benchmark	battery-casing	battery-exception	temperature-casing
examine scripts	TOP/bcasing.sh	TOP/bexception.sh	TOP/tcasing.sh
code dir	BN/bcasing	BN/bexception	BN/tcasing
data & run dir	BN/bcasing_run	BN/bexception_run	BN/tcasing_run
data file	BN/bcasing_run/run_ld_hc.txt	BN/bexception_run/run_ld_hcp.txt	BN/tcasing_run/run_ld_tent.txt
switch scripts	TOP/switch_to_bcasing.sh	TOP/switch_to_bexception.sh	TOP/switch_to_tcasing.sh

Figure 1: Directory structure organization summary. We assume TOP points to the untared benchmark suite, and BN points to a specific benchmark directory. For example, BN could be TOP/sunflow

input size	full_throttle boot	managed boot	energy_saver boot
large (ld)	run_ld_hc.txt	run_ld_mc.txt	run_ld_lc.txt
medium (md)	run_md_hc.txt	run_md_mc.txt	run_md_lc.txt
small (sd)	run_sd_hc.txt	run_sd_mc.txt	run_sd_lc.txt

Figure 2: battery-casing data file summary for the three boot mode contexts described in the paper across three different input sizes.

workload mode	full_throttle ent	full_throttle ent silent
full_throttle (ld)	run_ld_hcp.txt	run_ld_hcu.txt
managed (md)	run_md_hcp.txt	run_md_hcu.txt
energy_saver (sd)	run_sd_hcp.txt	run_sd_hcu.txt

Figure 3: battery-exception data file summary indicating how ent and silent ent runs are named across different workload mode sizes.

workload mode	ent run	java run
file	run_ld_tent.txt	run_ld_tjava.txt

Figure 4: temperate-casing data file summary indicating how ent and java temperature series data are named.

3. Examining Paper Data

You may inspect the battery-casing and battery-exception data using `TOP/bcaseing.rb` and `TOP/bexception.rb`. In addition, passing the scripts the -d or -p flag will display the data for Pi and Android benchmarks. Please note that `terminal-table` is required to use these scripts and we recommend installing via gems: `gem install terminal-table`. Please note that these scripts display the analyzed data discussed in our paper (energy difference, normalized energy saved etc). For a parsed dump of all raw data, you may add the -r option.

Examining Battery-Casing Data We will now highlight how to view the battery-casing data and how it corresponds to the data presented in our paper.

- **Running**: `./bcasing.rb`, with optional options `-p` and `-d` for Pi and Android data.
- **Script Output**: You will see the energy difference between managed and energy_saver boot runs compared against the full_throttle boot run for all input sizes (ld,md,sd), corresponding to the columns in the generated table. "Intel Raw Energy Difference" shows the raw difference between against the full_throttle boot mode run and "Intel Normalized Energy Difference" shows the difference normalized against the full_throttle boot mode run. In addition, raw plotting files are generated at `dat/bcasing.dat`, `pi_dat/bcasing.dat` and `droid_dat/bcasing.dat`.
- **Figure 6 Paper Data**: We present battery-casing runs for the large input sizes in Figure 6 of our paper. From the script, the "Intel Normalized Energy Difference" large data columns "`ld : managed boot`" and "`ld : saver boot`" are displayed as the numbers on the managed and energy_saver boot mode boxes respectively in Figure 6. `./bcasing.rb` will present System A's data, `./bcasing -p` will present System B's, and `./bcasing -d` will present System C's.

  Additionally, the raw plotting files contain the normalized data used to plot each box. The energy_saver and managed columns contain the energy consumed for those cases normalized against the full_throttle run as discussed in our paper.

Examining Battery-Exception Data We will now highlight how to view the battery-exception data and how it corresponds to the data presented in our paper.

- **Running**: `./bexception.rb`, with optional options `-p` and `-d` for Pi and Android data.
- **Script Output**: For `bexception.rb`, you will see the energy difference between ent and silent ent runs when an EnergyException is thrown under the following three

scenarios: energy_saver boot mode accessing managed workload mode, managed boot mode accessing full_throttle workload mode, energy_saver boot mode accessing full_throttle workload mode, corresponding to the three columns in the generated table. "Intel Raw Energy Difference" shows the raw difference between the ent and silent runs for each case, with ent and silent columns appended to show the individual runs for comparison purposes. "Intel Normalized Energy Difference" shows the difference normalized against the silent run column. In addition, raw plotting files are generated as `dat/bexception_BENCH_consumed.dat` where BENCH is the name of an Intel benchmark.

- **Figure 4 Paper Data**: We present battery-exception runs for all cases in in Figure 4 of our paper. Our script generates the data to be graphed and dumps it as dat files under `TOP/dat` for each benchmark. For example, `TOP/dat/bexception_sunflow_consumed.dat` contains the plottable data for Figure 4 sunflow. The data column corresponds to the workload modes, the context column corresponds to the boot modes, and the energy column represents the raw energy measurement.
- **Figure 5 Paper Data**: We present battery-exception runs where an `EnergyException` is thrown, in Figure 5 of our paper. From the script, the "Intel Normalized Energy Difference (silent runs)" columns "`saver boot - managed data`", "`managed boot : full data`", and "`saver boot - full data`" represent the ent boot mode boxes respectively. `./bexception.rb` will present System A's data, `./bexception -p` will present System B's, and `./bexception -d` will present System C's.

  Additionally, we have scripts to generate the normalized plottable data for Figure 5:
  ```
  ./bexception_all.rb
  ./pi_bexception_all.rb
  ./droid_bexception_all.rb
  ```
  will generate raw plottable files at

  ```
  dat/bexception_consumed.dat
  pi_dat/bexception_consumed.dat
  droid_dat/bexception_consumed.dat
  ```
  Here the data column corresponds to the workload mode, ent_managed and java_managed correspond to the managed boot and managed boot silent runs, and ent_saver and java_saver correspond to the energy_saver boot and energy_saver boot silent runs where each row represents one of the overlapping bars seen in the figure, i.e., the first row for a benchmark represents the first overlapping plot for a benchmark in Figure 5, the second row represents the second, and the third represents the third. The percent saved corresponds to the numbers displayed in `bexception.rb`.

Examining Temperature-Casing Data We will now highlight how to view the temperature-casing data and how it corresponds to the data presented in our paper.

- **Running**: `./tcasing.rb`
- **Script Output**: Generated dat files will for each benchmarks temperature-casing runs in TOP/dat dumped as a time series between ent and java runs. For example, TOP/dat/tcasing_sunflow_temps.dat contains the time series for sunflow's ent and java runs.
- **Figure 7 Paper Data**: These time series are plotted for each benchmark in Figure 7.

4. The Compiler and Runtime

The compiler is contained in the attached tar. Please untar it at a location of your choice. Specific directions for compiling and running ENT code are contained in the directory's README. We require that you set an environment variable named `ENT_HOME` to this path. In addition, please add `ENT_HOME/bin` to your path.

The Test Suite ENT contains a test suite located at `ENT_HOME/tests` that consists of compile checks and short generated code checks.

The compile checks are built off of Polyglot's test harness (`pth`) and may be viewed by examining `ENT_HOME/tests/pthScript`. `pth` checks for either a successful compilation, or inspects a compilation output for expected errors. Our tests are organized in to individual folders.

The runtime tests use a script `run.rb` that will compile the ENT code in a folder and then run the code and either expect a successful run, or an exception to be thrown. Typically, these runtime tests are contained in directories labeled 'pass' or 'fail', and have a file called `TEST` that helps `run.rb` run. You may execute a specific test by `./run.rb snapshot_pass1` etc.

The entire test suite may be launched from `test.bash` by simply executing the script. It will execute all static tests and all runtime tests and report the results.

Further information is contained in `ENT_HOME/tests/README`.

In addition, we have provided the code examples used in the paper as compilable code stubs in the `tests` directory. They are contained in subdirs `paper_ex1`, `paper_ex2`, and `paper_ex3` for Listing 1, 2 and 3 respectively. You may compile each by: `entc -d build paper_ex1/*.java` etc.

5. Intel Benchmarks

System Setup We used an Intel i5 CPU laptop with 4GB RAM, Ubuntu 14.04 LTS OS, and Java 1.8 JVM. The specific model is a lenovo X230.

Running The details of each benchmark, scripts, directories, and data files are all listed in Figure 1. Applications

consistent of one or more of the benchmarks (battery-casing, battery-exception, and temperature-casing) discussed in the paper. Each benchmark has an associated data directory which we have already discussed, and code directory. Each application has several scripts to switch between benchmarks that active code and compile it. These are `switch_to_bcasing.sh`, `switch_to_bexception.sh`, and `switch_to_tcasing.sh`.

To gather energy measurements, load the msr module via: `sudo modprobe msr`. As accessing the msr module requires root privileges, you will need to execute the run scripts as root in order to gather energy readings. We recommend adding `alias sudo='sudo env PATH=$PATH JAVA_PATH=$JAVA_PATH'` to your `.bashrc` to fix path issues when using `sudo`. Note that you may still run the benchmarks without loading the module or using `sudo` — energy readings will simply fail and you are likely to see pread errors.

We have provided high level scripts to switch all applications to a specific benchmark and run a single instance for each to demonstrate functionality. We recommend beginning here without using `sudo` as it will affect compilation. They are `TOP/intel_bcasing.sh`, `TOP/intel_bexception.sh`, and `TOP/intel_tcasing.sh` for battery-casing, battery-exception, and temperature-casing respectively.

The following is a step by step guide to run the demonstration scripts. Please note that these will not reproduce results, and are instead meant to test the benchmarks on your system. We assume `TOP` is the current working directory.

- 1. `sudo modprobe msr`
- 2. `./intel_bcasing.sh` or `./intel_bexception.sh`, `./intel_tcasing.sh`

At this point, results may be reproduced as discussed in the following section.

5.1 Reproducing Results

The data directory also serves as the run directory where you will find `all.sh` which will run ALL instances of the benchmark, i.e., all input sizes, all boot mode cases, and multiple iterations of each. This may take time and will overwrite the recorded data contained in the directory.

Reproducing Battery-Casing Results The following is a step by step guide to gather results for `sunflow` battery-casing, as well as view the data. We assume `TOP` is the current working directory.

- 0. `sudo modprobe msr`
- 1. `cd sunflow`
- 2. `./switch_to_bcasing.sh`
- 3. `cd bcasing_run`
- 4. `sudo ./all.sh`
- At this point, new data files will be generated for each case as described in 2.
- 5. `cd $TOP`
- 6. `./bcasing.rb`

Reproducing Battery-Exception Results The following is a step by step guide to gather results for `sunflow` battery-exception, as well as view the data. We assume `TOP` is the current working directory.

- 0. `sudo modprobe msr`
- 1. `cd sunflow`
- 2. `./switch_to_bexception.sh`
- 3. `cd bexception_run`
- 4. `sudo ./all.sh`
- At this point, new data files will be generated for each case as described in 3.
- 5. `cd $TOP`
- 6. `./bexception.rb`
- At this point, you will see updated data for the percentages for `sunflow` for Figure 6. Additionally, `/dat/bexception_sunflow_consumed.dat` will contain updated data for Figure 5.

Reproducing Temperature-Casing Results The following is a step by step guide to gather results for `sunflow` temperatue-casing, as well as view the data. We assume `TOP` is the current working directory.

- 0. `sudo modprobe msr`
- 1. `cd sunflow`
- 2. `./switch_to_tcasing.sh`
- 3. `cd tcasing_run`
- 4. `sudo ./full.sh`
- At this point, new data files will be generated for each case as described in 4.
- 5. `cd $TOP`
- 6. `./tcasing.rb`
- At this point an updated time series will be generated in `/dat/tcasing_sunflow_consumed.dat` and will contain updated data for Figure 7.

6. Wattsup Logging

Pi and Android energy measurements are collected by using a Wattsup power meter which logs energy consumed in watts at 1 second intervals. Both the Pi and Android devices benchmark's output timestamps that indicates the start and end of an individual run and require an external script to synch this data with data logged from the meter.

We have modified the source code for a logging utility for the meter and have included this in `TOP/Watts-up--logger`.

pyserial is required to run the logger over usb; we recommend installing via pip.

You may start the logging script by ./wattsup -l -o out.log which will continue logging until the user presses Q. During this time you may run as many benchmarks that produce timestamp files as you wish. Afterwards, you may create energy files from the log and timestamp data using TOP/timestamp.rb, i.e., ./timestamp.rb out.log stamp.txt. We discuss specifics of Pi and Android data collection below.

7. Pi Benchmarks

System Setup We used a Raspberry Pi 2 Model B with 1GB RAM, Raspbian Jessie OS, and Java 1.8 JVM, with a keyboard, mouse, HDMI monitor, and ethernet cable connected. In addition, we installed the Pi camera module.

Running Due to having to execute Pi code on a separate machine there is a slight variation of directory structure. TOP/pi_bench contains the timestamps and data collected for pi experiments with the same layout in Figure 1 with the exception the code and run scripts are located in TOP/pi_bench_code, but follow the same directory layout. If you would like to run pi benchmarks, please copy TOP/pi_bench_code to the device, along with an installed version of ent. Note that you will have to setup the paths on the pi as you did in Intel.

The script layout is the same for Pi and Intel, with the exception of the top level kickoff scripts which we have named pi_bcasing.sh and pi_bexception.sh. One notable difference is that the pi runs will not generate energy files; timestamp files are created instead. These need to be synched according to the instructions in Section 6. Note that root is not required as no energy measurements come directly from the device.

8. Android Benchmarks

System Setup We used a Google Nexus 5X, Android 6.0 Marshmellow OS, Android Runtime (ART) 2.1.0. Please make sure the Android SDK installed on your system for building the applications and accessing the device through the Android Debug Bridge (adb). The following Android SDK versions are required: 22, 23, and 25. The following build-tool versions are required: 22.0.1, 23.0.2, and 23.0.3. The 'Android Support Repository', 'Google Repository', and 'Google Play services' are also required, and can be installed from the Android SDK manager as well. The authors are happy to assist if there is an issue.

Please rebuild the jrapl-port using Java 7, and then rebuild ENT using Java 8. After this step you are free to compile and link together the Android packages.

Running We will refer to the top level Android benchmark directory (TOP/android_benchmark) as DTOP. Running Android benchmarks requires the RERAN framework. Furthermore, actually executing the Android benchmarks requires a running shell (adb shell) on the device to kick things off. We have provided scripts that will create a directory structure and load all replays and run scripts to the device.

First, please add TOP/android-scripts to your path. Then, with the device connected, execute DTOP/load-dir.sh and then DTOP/load-dat.sh. This mirrors the directory structure from the benchmark suite on to the device. You can expect similar steps as found in Intel and pi with the additional step of starting the benchmark from the device. In the run directories you will find run.sh which functions as all.sh does on Intel and pi, and qrun.sh which will kickoff a single iteration of all cases for a benchmark.

The following is a step by step procedure to start NewPipe.

- 1. (on linux) cd DTOP/NewPipe
- 2. (on linux) ./switch_to_bcasing.sh
- 3. (on device) cd /data/local/tmp/newpipe/bcasing_run
- 4. (on device) ./qrun.sh

Please note that the replay logs are sensitive: NewPipe requires the device be in landscape mode with the usb port facing the left; please run all other apps in portrait mode.

To collect data, plug the device fully charged into the power meter. As this requires the device's usb port to be plugged into the charger, scripts will have to be executed over the wireless debug bridge. When the script finishes, execute pull.sh in the run directory on linux to pull the timestamp data from the device.

A few app specific notes. Please execute DTOP/duckduckgo/make-prop.sh before building duckduckgo. This script assumes you have set ANDROID_HOME to your Android SDK. We also require a dependency for MaterialLife: Please enter DTOP/MaterialLife/rainbow and execute ./gradlew install before building MaterialLife.