This is a cookbook planned for developers or researchers that want to use CT+, a tool created to improve the energy-efficiency of Java applications by optimizing the usage of Java collections.

More information can be found on our website:

https://energycollections.github.io

1 - Configure dacapo+JRAPL

First it will be necessary to download the most recent version of dacapo (https://github.com/dacapobench/dacapobench/tree/dev-chopin), or download the version we used, already configured, available on our website (https://energycollections.github.io/)
As today (19/03/2020) the most recent version of dacapo has a dependence problem when trying to access the maven repository. This problem is solved in the configured version.
To collect the energy of the dacapo benchmarks we use JRAPL through a callback. The necessary files can be found in the website on the "configuration" section.
It's important to notice that because we are using RAPL, the solution will only work on unix systems.
Although dacapo offers support for different versions of JDK, in our work we used the **JDK8** for every experiment. Other versions may not work as intended. **Ant** it's also necessary. In this work we used the ant version 1.10.7.

To build a dacapo benchmark the following command is necessary:

1. Build the benchmark that it's gonna be executed using the following command inside the folder `/dacapo/benchmarks/`:

```
ant "benchmark" (e.g., ant cassandra)
```

2 - Using the CT+

Our tool CT+ is divided in three different modules: the profiler, the analyzer and the recommender. Before using the recommender, it's necessary to create the device's profile and analyze the application that will be modified.

2.1 - Creating the profile

To create the device's energy profile it will be necessary to create a new Java project using the source code of our profilers (CT+ Desktop Profiler or CT+ Android Profiler), they can be found on the project website.

Because it's necessary to collect the energy, the profiler can be only created on unix systems.

To create the energy profile the following steps and commands are necessary:

1. Execute the following command to allow the RAPL to collect the energy data:

   `modprobe msr`

2. Execute your IDE using root access
3. Run the project as application using the class `MainTest`
4. Choose to execute the class with parameters (e.g., on Eclipse, right click on the class and choose 'Run As' - 'Run Configurations')
5. Choose the option to pass arguments for the execution of the class. The pattern for the arguments are as follows:

   `[hash|list|set] #threads amount_of_operations capacity loadfactor`

   `(e.g., set 4 175000 175000 0.75)`

   Parameters up to capacity are used by lists and sets, maps also use the fourth parameter, which is a float between 0 and 1. In our work, we used a `loadfactor` of 0.75. For unsafe collections, only one thread will be used at any time. It's not possible to predict an exact value for the capacity. In our work, we used three different capacities to exercise different scenarios.

6. The output will be printed in the console as follows:

```
set
NumberOfThread,Frequency,power_limit(pkg-dram),time_window(pkg-dram),Metho
dName,WallClockTime,CpuTime,UserModeTime,KernelModeTime,DramEnergy0,CPUEne
rgy0,PackageEnergy0,DramPower0,CPUPower0,PackagePower0,DramEnergySD0,CPUEn
ergySD0,PackageEnergySD0,WallClockTimeSD
Conf: Iterations=10, threads=4, N=175000, capacity=175000
synchronizedLinkedHashSet,add(value),0.10152,0.47323,0.64878,0.0
synchronizedLinkedHashSet,iterator,0.00479,0.01774,0.02489,0.0
synchronizedLinkedHashSet,remove(key),0.16644,0.89954,1.16577,0.0
synchronizedHashSet,add(value),0.09999,0.55496,0.73115,0.0
synchronizedHashSet,iterator,0.00672,0.034,0.04595,0.0
synchronizedHashSet,remove(key),0.19574,1.00061,1.28851,0.0
```

   The highlighted part represents the consumption of the package and will be used to create the energy profile. The other values will not be used.

7. Collect all the data from each different group of collection (hash, list and set) and organize it in the following format:

Collection	Operation	Consumption
synchronizedLinkedList	add(value)	0.21
synchronizedLinkedList	iterator	0.07
synchronizedLinkedList	sequentialGet	308.9
synchronizedLinkedList	randomGet	774.95
(...)	(...)	(...)

   It's very important that the header should be exactly as shown in the table above (Collection - Operation - Consumption). The order where each collection is present in the table it's not relevant. All the data should be saved in the .csv format. An example of energy-profile can be found here.

If necessary, the energy output can be more precise. To adjust the precision, go to the class `DataPrinter`, on the method `printResult` and adjust the `DecimalFormat df` as required.

2.2 - Analyzing the application

To analyze the application the following steps and commands are necessary:

1. Select the .jar that you want to analyze. In the specific case of dacapo, there's several different .jar's that are used for the different benchmarks. Usually, there's a .jar that it's used while executing the benchmark that can be found on the .cnf file. Using the benchmark `biojava` as an example:

   `dacapo/benchmarks/bms/biojava/biojava.cnf:`

   ```
   "benchmark biojava
     class org.dacapo.harness.BioJava
     thread-model per_cpu
     jars "AAProperties-jar-with-dependencies.jar";
   (...)"
   ```

In the example above, `AAProperties-jar-with-dependencies.jar` is used. That .jar can be found on the following path:

`dacapo/benchmarks/bms/biojava/build/biojava-aa-prop/target/`

Every benchmark has a different path for a different .jar. The site of the project gives more details about the selected .jar's.

2. The command to execute the analyzer is the following:

   `java -jar CECOtool-analyzer.jar -t "path/to/selected.jar" --packages "com.my.package" --analysis-output-file "my-jar-analysis.csv" -a`

   `e.g.,`

   `java -jar CECOtool-analyzer.jar -t "AAProperties-jar-with-dependencies.jar" --packages "org.biojava" --analysis-output-file "analysis-biojava.csv") -a`

   The parameter -a specifies that this is an analysis, -t directs for the target .jar. The package is an optional parameter in case the user doesn't want to analyze the full application. It can be useful to focus on the packages that represent the application's core.

2.2 - Selecting the recommendations

To create the file with recommendations, the following steps and commands are necessary:

1. Both the energy profile and the analysis are necessary to make the recommendations (e.g., `energy-profile.csv` and `analysis.biojava.csv`)
2. The command to create the recommendation file is the following:

```
java -jar CECOtool-analyzer.jar --analysis-output-file "my-jar-analysis.csv"
          --energy-profile-file "path/to/my/energy-profile.csv"
        --recommendation-output-file="recommendation-file.csv" -r
```

   The parameter -r specifies that this is a recommendation. The recommendations will be placed on the file specified on the parameter --recommendation-output-file.

2.3 - Applying the recommendations

To apply the recommendations, the following steps and commands are necessary:

1. Go to the build directory of the selected benchmark and find the necessary files. Sometimes there's a folder with the project files (e.g., `cassandra`) or all the files are already inside the build folder (e.g., `graphchi`). For the following steps, the benchmark `graphchi` will be used as an example.
2. Copy the necessary files to another location (e.g., `/my-benchmarks/graphchi/build`). If the files were direct the folder build, create a folder build to put the files. If the files were inside another folder, just copy the entire folder (e.g., `/my-benchmarks/cassandra-3.11.5`). The folder's name of the original source files should be the same as inside the build folder. Copy the original source code and modified the name of the folder (e.g., `/my-benchmarks/graphchi/new-build` or `/my-benchmarks/cassandra-3.11.5-modified`).
3. Some modifications can result in small bugs that could be solved with a deeper understanding of the system function. Because we want to keep the modification as simple as possible we'll just discard those modifications. To ease that task, inside the folder that will be modified, use the following git commands:

   ```
   git init
   git add -A
   git commit -m "original state"
   ```

   Although that step it's not necessary for the full execution of CT+, it's deeply recommended.
4. To apply the recommendations the following command needs to be executed:

   ```
   java -jar cecotool-transformer.jar -f="recommendation-file.csv"
                   -t="path/to/source/until/java"
   ```

   e.g.,

   ```
   java -jar cecotool-transformer.jar -f="rec-graphchi.csv"
   -t="/home/user/my-benchmarks/graphchi/new-build/src/main/java"
   ```

   This command will apply the modifications to the source code. In the case of some bug when applying the modification, it will be necessary to remove the bugged collection recommendation from the recommendation file. In the actual version of CT+, this needs to be done manually but in future versions this will be done automatically by the tool.

2.4 - Creating a patch file

To create and use a patch file, the following steps and commands are necessary:

1. Execute the following command:

   diff -ru -w -u0 --exclude=".git" path/to/original/folder path/to/modified/folder > additional.patch

   In the case of the command been executed inside the folder with original and modified versions (e.g., `/my-benchmarks/graphchi` with `/my-benchmarks/graphchi/build` and `/my-benchmarks/graphchi/new-build`), just the folder name is necessary

   diff -ru -w -u0 --exclude=".git" build new-build > additional.patch

2. It's important to notice that the name of the original folder needs to be exactly the same as the folder found inside the build folder of dacapo (or in the case of the build, just the name build)

2.5 - Applying the patches

To apply a patch file in the modified version of dacapo, the following steps and commands are necessary:

1. Copy the .patch file you created to the benchmark folder (e.g., `dacapo/benchmarks/bms/$benchmark/additional.patch`)
2. Modify the following lines on the `build.xml` file inside the benchmark folder.

   `dacapo/benchmarks/bms/$benchmark/build.xml:`

   ```
   <patch patchfile="${bm-files}/additional.patch" dir="${bm-build-dir}" strip="0" ignorewhitespace="yes"/>
   ```

   Change the highlighted part to the patch file you created.

3. To apply a patch, just build the selected benchmark with the following command on the benchmarks folder on dacapo (`/dacapo/benchmarks/`):

   ant "benchmark" (e.g., ant cassandra)

3 - Execute dacapo+JRAPL

1. Execute the following command to allow the RAPL to collect the energy data:

   `modprobe msr`

2. Inside the benchmarks folder on dacapo (`/dacapo/benchmarks/`), execute the benchmark using the following command:

   `java -jar dacapo-vers.jar -n x -s y -c EnergyConsumptionCallback “benchmark”`

   The “dacapo-vers” change based on the version of dacapo being executed. The parameters -n and -s define, respectively how many times the benchmark will be executed and the size of the workload. On our work, we executed each benchmark 70 times with the default size (i.e., `-n 70 -s default`) with the exception of tomcat in which we used the large workload (i.e., `-s large`)

3. Each time the benchmark is executed you will need to build it again to apply a different patch. If you want to remove the applied patch, just comment the line where the patch is supposed to be applied and build the benchmark again:

   `dacapo/benchmarks/bms/$benchmark/build.xml:`

   ```
   <!--patch patchfile="${bm-files}/additional.patch" dir="${bm-build-dir}" strip="0" ignorewhitespace="yes"/-->
   ```

   `dacapo/benchmarks:`

   `ant “benchmark”`

As an optional step, in our study, we executed the command **`script “filename” (e.g., script tomcat-modified-1.txt)`** before step 2. The reason behind it is to ease the work of collecting the results from the terminal. After the execution, the command **`exit`** is necessary to save the file.

Step 2 will print the results from the execution on the terminal, including the time and three different consumptions. For our study, we used only the `package` consumption. Here’s a example of an execution:

```
===== DaCapo 9.12-MR1-git+unknown tomcat starting warmup 2 =====
Loading web application
Creating client threads
Waiting for clients to complete
Client threads complete ... unloading web application
854,804871#2875,757629#5022,038086
===== DaCapo 9.12-MR1-git+unknown tomcat completed warmup 2 in 11660 msec
=====
DRAM consumption: 17.57666100000006
CPU consumption: 74.134094
PACKAGE consumption: 105.72595199999978
854,898315#2876,253357#5022,736694
```