Atom-LEAP

User-space Measurement Tutorial

1 Introduction

The Atom-LEAP platform is a highly-instrumented Intel Atom based motherboard along side a USB-enabled National Instruments data acquisition module (DAQ). It allows researchers to acquire power measurements at a high granularity, in both kernel and user-level code. This brief tutorial will describe how to use the Atom-LEAP to acquire user-level data using the core LEAP tools as well as the more user-friendly LEAPfrog (LEAP For Repetitive, Organized Gathering) package.

There are three main steps in the data gathering process on the Atom-LEAP. First, data acquisition involves starting the sampling device and directing output to a log file. Workload execution begins once sampling has started properly. Finally, reporting takes the raw data and timestamps generated in the first two steps and generates useful reports. We will first describe the use of the core tools in acquiring and processing data from the Atom-LEAP, and then go on to describe use of the more user-friendly LEAPfrog.

2 What does it measure?

The Atom-LEAP samples data during the runtime of a specified process. The sampling hardware observes the watts (W) consumed by a component or components on a particular circuit during a sampling interval. This interval is typically 1/10000th of a second, or 100 microseconds. Because we have the watts and the run time, we can calculate the total joules (J) during the observed task.

The Atom-LEAP samples data for 5 components by default, although it can be configured to acquire more. Those components are: CPU, hard disk, bridge (experimental), RAM, and USB. In all cases, the measurement is taken on the pins or wires responsible for powering the device. This means, for example, that the hard disk power measurement is on the power leads to the disk itself, not to the disk controller. Likewise, the RAM power is measured directly at the DIMM, the CPU at its power converter, and the USB on the USB cable. The bridge measurement is currently experimental.

When the sampling is finished for a task, the raw data exists as rows of watt measurements per component, per sampling interval. Synchronization is performed to identify the system time when sampling began and ended, as well as to identify the specific segments of samples observed relevant to the workload or workloads under investigation.

3 Core process

3.1 Data acquisition

Experimental data is acquired by a USB-enabled National Instruments data acquisition module (DAQ). The DAQ is connected to current-sensing leads which are attached to various components of the LEAP node. Typically, the DAQ samples data at 10 kHz, or 10,000 times per second. In order to acquire any data, the DAQ must be instructed to start gathering data, and must be given an output location. Finally, the sampling process must be run at real-time priority, so that it is not interrupted by other processes.

3.1.1 Load kernel module 'probe.ko'

The kernel requires a module in order to interact with the DAQ. This probe must be loaded the first time the Atom LEAP is booted (or when the DAQ is connected to the machine). To load the kernel probe, run:

```
$ sudo insmod /usr/atom_LEAP/code/sync/probe.ko
```

The probe will add messages to `/var/log/messages` which will enable us to measure the amount of time which passed during the sampled workload(s).

3.1.2 Start the sampling process

If the kernel module is loaded, you can start the sampler by opening a terminal and executing:

```
$ sudo chrt 99 start_sampling 2> data.txt
```

Sampling is started using `sudo` for two reasons. First, the sampling process must interact with the kernel `probe` module, which requires root privileges. Second, the sampler must be set to run at real-time priority so that the sampler program (`start_sampling`) does not fall too far behind the physical DAQ device. When the sampler first starts, it attempts to synchronize with the system. It is important to wait for synchronization *before* starting your workload. (The accuracy of your data will not be affected by this time gap due to the synchronization code.)

Either of the two following messages indicates that the sampler is ready:

```
[45177.536016] Synchronized! Core = 0 TSC = NNNNNNNNNNNNN
[45177.577051] Probe Unregistered
```

...or

```
[32383.559476] KProbe address set to c025d560
[32383.681772] Probe Registered
```

Once you see this output, you're set to run your workload!

3.1.3 Sampling tips

3.1.3.1 Don't run the sampler until your workload is ready.

You generally don't want to run the sampler until you are ready to run your workload. This is because the sampler generates a large amount of data. The longer the sampler runs, the longer it will take to synchronize the data and generate reports. If you've started the sampler early and you want to quit, press `^C` to stop it.

3.1.3.2 What if the sampler doesn't become ready?

Sometimes the sampler never prints either of the two lines of data, or prints some errors. In this case, just press `^C` to kill the sampler, wait a few seconds, and try again.

3.2 Preparing and executing your workload

The workload process is simply a task from which you would like to obtain power readings. This workload can be in the form of a compiled program, a shell script, or some combination of the two. We'll use a compiled C program as an example for the core tools, and discuss more complex workloads later.

Before your code can be profiled, you must identify the portions of code for which you want to collect data. For each such section, you'll need to insert a small amount of instrumentation code into your program.

3.2.1 Energy Calipers

3.2.1.1 A familiar example

Suppose you wanted to monitor the power consumption of the following *somewhat* familiar program:

```
#include <stdio.h>
main()
{
        int a = 0;
        for (a; a < 100000; a++) {
                printf( “hello, world  “);
        }
}
```

In order to acquire accurate data, we need to know precisely when our workload begins and ends.

3.2.1.2 Using `getticks()` to mark your workload

If the measurement is to be accurate, we need to know exactly when the workload process starts, and when it ends. To do that, we read the value of the Atom CPU's timestamp counter (TSC). This is done by adding a function named `getticks()`. `getticks()` simply runs some in-line assembly code to read the timestamp counter and return it.

After we read the TSC with `getticks()`, we are ready to execute our workload. This workload could be any C code, or external programs executed with the `system()` library call. After the work to be measured is complete, we read the TSC with another call to `getticks()`. These start and end timestamps become the workload’s bookends, or “energy calipers” -- marking the points in the sampled data that correspond to the workload code we just executed.

In order to use this data after the program completes, we need to log this information in a control file that the data processing tools can read. The control file format is:

```
MULTIWORD_DESCRIPTION, TAG, START_TIME, END_TIME
```

The multi-word description is meant to be short, human-readable description of the workload.

The “tag” is a one-word identifier for the portion of code being measured (see “tabular output” below), and the start and end times are values from `getticks()`. Putting it all together, this looks like:

```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned long long ticks;

/* time stamp function */
static __inline__ ticks getticks(void)
{
        unsigned a ,d;
        asm("cpuid");                                   /* Serialize */
        asm volatile ("rdtsc": "=a" (a), "=d" (d));     /* Read TSC */

        return (((ticks)a) | (((ticks)d) << 32));       /* 64-bit TSC */
};


main()
{
         ticks TSC_start, TSC_end;
         int a = 0;
         char *label =  “hello” ;

         TSC_start = getticks();    // get start time

         for (a; a < 100000; a++) {
                 printf( “hello, world ” );
         }

         TSC_end = getticks();              // get end time

         // write out timestamps and labels
         FILE *caliper_file = fopen("/ramdisk/Energy_Caliper_Control_File","w");
         fprintf(caliper_file, "This is a test!,%s,%llu,%llu¥n", label, TSC_start, TSC_end);
         close(FILE);

}
```

As you can see, the program is essentially the same – we’ve simply added the `getticks()` function, read the TSC before and after the workload, and wrote a log-line out to a control file when everything was finished. The control file is written to a ramdisk so as not to induce extra workload on the hard disk.

3.2.1.3 Using multiple calipers to measure workload components

What if we want to measure discrete parts of code within a larger task? For example, when profiling a hypothetical compression utility, we might want to separately measure the power consumption associated with reading the file, compressing the file, and writing the file. This can be done trivially by adding “energy calipers” around all three sections of code. It is easier to analyze the data if the calipers do not overlap, but that is not strictly necessary. The only requirement is that the energy caliper control file has a line for each measured section of code.

3.2.1.4 Beware of optimization!

As anyone who has spent much time profiling C code can tell you, both your instrumentation and your workload code can be unexpectedly re-ordered or optimized by the compiler in such a way that it no longer measures what you intend. In particular, if your calls to `getticks()` are both placed after your workload, you won't collect the correct data. It's probably a good idea to test your workloads with different levels of optimization as a sanity check.

3.2.2 Executing your workload

Once your workload is prepared, start the sampler as described above. When it the sampler is ready, run your workload executable in a different terminal like so:

```
$ sudo chrt 99 ./megahello
```

Remember that the Atom LEAP is not simply measuring runtime – it is measuring the energy dissipated by discrete components. Your workload may affect this in unexpected ways. Consider what you are trying to measure, and how you could best execute that workload in isolation – you may want to redirect the output of your program to a file, use a ramdisk instead of a disk for output, or send output to `/dev/null`.

3.2.3 Stopping the sampler

Once your workload has completed, you should wait 15 seconds in order to let the sampler collect all relevant data. Then, stop the sampling process by pressing `^C` once (or twice) in the terminal where the sampler is running. You have just created two artifacts: the sample file `data.txt`, and the control file holding your timestamps, which should be located in `/ramdisk/Energy_Caliper_Control_File`. Additionally, the kernel log `/var/log/messages` contains important information for synchronization.

3.3 Processing your data

Processing collected data consists of two main steps. First, the sampled data in the file `data.txt` must be synchronized with the kernel timestamps in `/var/log/messages`. Second, the control file and synchronized data file are examined, and summaries are generated for each set of energy calipers listed in the control file.

3.3.1 Synchronizing the data

To synchronize the data, simply run:

```
$ sync.py data.txt > sync-data.txt
```

This will synchronize kernel events with the timestamps listed in your `data.txt` file. Now, all we need to do is generate a report from the synchronized data.

3.3.2 Generating a report

In the simplest case, a report can be generated from your synchronized data by running:

```
$ user_caliper_report.py sync-data.txt
```

The reporting tool will automatically open `/ramdisk/Energy_Caliper_Control_File` and use it to extract and summarize average statistics for each time span listed in the control file. The statistics are printed in this format:

```
print hello world 100k times (hello)

CPU_Energy (J) = 8.669092
HDD_Energy (J) = 6.065411
Bridge_Energy(J) = 1.179657
RAM_Energy (J) = 2.988264
CPU_Average_Power (W) = 2.665445
HDD_Average_Power (W) = 1.864903
Bridge_Average_Power(W) = 0.362703
RAM_Average_Power (W) = 0.918787
```

In the above example, "print hello world 100k times" is the message in the control file. The word "hello" is the tag associated with this timespan, and while we do not see the TSC_start and TSC_end, we see the average energy (in joules) and power (in watts) consumed during the time between those timestamp values. If you included multiple sets of timestamps, you would see one stanza for each set of timestamps. Congratulations – you just acquired high-granularity power data from the Atom LEAP!

3.4 Data Analysis and Workload Design

3.4.1 Watts vs. Joules

Data is printed in both watts and joules. The watts value shows the average instantaneous power being consumed by each component over the entire workload. The joules values show the amount of power consumed *over the runtime of the workload* for each component. For example, in the "hello, world" example, we see that the CPU consumed 2.67 watts (on average), while the hard disk consumed 1.86 watts. However, the CPU is said to have consumed 8.67 joules during the workload while the disk consumed 6.06 joules. By comparing the ratio of joules to watts for each workload, we can see that:

8.67/2.67 ~= 6.06/1.86 ~= 3.25 seconds

It is important to remember that the same *functional workload* – that is, code which accomplishes the same result – will not necessarily have the same runtime, nor will it necssarily draw the same amount of power from the components depending on the operation of the components under the implementation of the workload. Additionally, the runtime and reported data will vary slightly due to noise and interference. All these issues highlight the importance of careful workload design.

3.4.2 Workload Design

Any reasonable testing framework makes it straightforward to compare the performance of program A using algorithm X versus algorithm Y. However, systems testing is rarely that simple. While sshfs and Samba over an encrypted tunnel may accomplish the same outcome, the code they execute – and how they execute it – is very different. It is not simply a matter of pulling out one algorithm and sticking in another.

As a simple example, `sshfs` essentially invokes `sftp` processes for each file operation. In contrast, Samba is a fully-featured networked file system managed by several kernel components. In addition, the use of an `ssh` tunnel makes Samba dependent on a separate user-level process – meaning that packets sent over the tunnel have an additional penalty associated with the context switch and execution of that process.

For reasons like these, it is absolutely critical that you consider all the components of your testing environment, and make sure that you have streamlined your workload as much as possible.

4 LEAPfrog

With this background material behind us, we are ready to discuss the LEAPfrog package, which automates the core process and makes it easier to acquire and process data in a format suitable for statistical analysis.

LEAPfrog is a package of bash shell scripts meant to automate interactions with the core Atom LEAP tools. LEAPfrog aims to do several things:

1. Facilitate instrumenting shell scripts and interpreted code
2. Modularize workloads and testing software
3. Automate repetitions of workloads
4. Automate starting and stopping the sampler
5. Automate generating reports
6. Generate output in an analysis and backup-friendly format

We'll talk about each of these topics in the following sections, and show you how to use the LEAPfrog package to create your own workloads without compiling C code.

4.1 Workloads

4.1.1 Instrumenting modular shell scripts

While compiled C code is perfect for measuring the power consumption of algorithms, it is not convenient for testing higher level tasks, especially the sequential, interrelated tasks common in systems testing. For example, suppose we would like to measure the power consumption involved in decompressing a tar file and then deleting the results of the decompression. We could do this in C, using multiple `system()` calls, but this requires recompilation for any changes. However, the C example above gives us a format we can translate to bash with the addition of a `getticks` binary.

```
#!/bin/bash

MESSAGE=" untar linux"
TAG=" untar"
TSC_START=`getticks`
tar xvjf linux.tar.bz2
TSC_END=`getticks`
echo  "$MESSAGE, $TAG, $TSC_START, $TSC_END"  > /ramdisk/Energy_Caliper_Control_File

MESSAGE=" remove linux"
TAG=" remove"
TSC_START=`getticks`
rm -rf linux-2.6/
TSC_END=`getticks`
echo  "$MESSAGE, $TAG, $TSC_START, $TSC_END"  >> /ramdisk/Energy_Caliper_Control_File
```

Bash scripts can also easily include “setup” and “teardown” portions to ensure that the experimental environment always starts in the same state. (That isn't shown here, although the example `/usr/atom_LEAP/code/frog/ext2untar-workload` includes code to do this.)

The Atom LEAP system provides a `getticks` binary, which prints the value of the TSC to standard out. With a `getticks` binary, we can perform the steps from the core process section using flexible interpreted code such as bash scripts to run our workloads (and add necessary lines to the control file). While we may lose some accuracy due to process startup costs, this should be insignificant for higher-level workloads like filesystem benchmarks and the like.

4.1.2 Automating repetitions

The performance characteristics of modern computer systems is probabilistic. Cache behavior, networking, process interference, pipelining, and hard disk behavior (just to name a few) all conspire to cast doubt on the representativity of any single measurement. Thus, when measuring systems, it is necessary to take several measurements of the same process and analyze them in order to make sure that the results are accurate.

We can do this very easily by enclosing the workload in a for loop:

```
#!/bin/bash

REPS=$1   # the first parameter is the number of repetitions

sudo chrt -p $$ 99 # set the priority of this process

source /usr/atom_LEAP/code/frog/frogscripts.sh # load the control_append function

for REP in $REPS
do

        MESSAGE=" untar linux #$REP"
        TAG=" untar"
        TSC_START=`getticks`
        tar xvjf linux.tar.bz2
        TSC_END=`getticks`
        control_append $TSC_START $TSC_END $TAG $MESSAGE

        MESSAGE=" remove linux #$REP"
        TAG=" remove"
        TSC_START=`getticks`
        rm -rf linux-2.6/
        TSC_END=`getticks`
        control_append $TSC_START $TSC_END $TAG $MESSAGE
done
```

4.1.3 Enhancing the workload

You may have noticed that we made a few small changes to this file. First, we added the line:

```
sudo chrt -p $$ 99
```

... which sets the process id (PID) of the workload process (and its children) to real time priority in order to minimize preemptions. This requires sudo access with the NOPASSWD option set. (See "sudo settings" below.)

The next change is that we've sourced the `frogscripts.sh` package in order to load the `control_append` bash function. As you can see, `control_append` is simply a shortcut for echoing the test information to the control file. Bash parameters are positional; it is the *order* of the arguments to `control_append`, not the variable names, that matter. Note that `control_append` reads the parameters in a different order; this is to simplify the code.

Thanks to these changes, executing:

```
$ workload 10
```

... will run the workload 10 times in a row at real-time priority, and the MESSAGE variables will be updated for each repetition with the $REP counter variable. While the MESSAGE variable is updated for each REP, the TAG variable is not. That is so that the data associated with each TAG can be separated for easier analysis.

4.2 Automating the sampler and reporting

As you read in the previous sections, starting and stopping the sampler requires a second terminal and patience, both while waiting for it to synchronize and to finish collecting data. LEAPfrog provides a tool, `leapfrog`, which takes all the hard work out of collecting data. `leapfrog` also sources `frogscripts.sh`, which includes a number of helper functions to:

- start the sampler in a subshell and wait for the appropriate messages
- run a given workload a number of repetitions
- stop the sampler when the workload is complete
- run the synchronization tool on the output
- run the reporting tools on the synchronized data
- back up the data
- attempt to behave intelligently when something goes wrong

`leapfrog` starts the sampler, calls the given workload with any arguments, and cleans up when finished. With a workload script like the example above, simply execute:

```
$ leapfrog workload 10
```

This will start the sampler, run the workload 10 times, stop the sampler, and perform the post-run data processing steps automatically, saving the data out into files.

4.3 LEAPfrog backup behavior

While the Atom LEAP is not suitable for multiple simultaneous experiments, researchers will want to run multiple, identical, tests in sequence to establish statistical confidence. It's common for users to reuse the same names for input and output files, but this can facilitate accidentally overwriting important information. For this reason, LEAPfrog copies all the relevant data files to `/tmp/UNIXDATE-workload/FILENAME` where `UNIXDATE` is the number of seconds since the UNIX epoch and the value of `FILENAME` is one of `data.txt`, `sync.txt`, `Energy_Caliper_Control_File`, or `reports` (a directory which containing the tabular output for each tagged workload subcomponent).

4.4 LEAPfrog tabular output format

LEAPfrog prints output to the console in a manner similar to the “core process” just described. However, because of the necessity of running multiple repetitions for statistical confidence, LEAPfrog also prints results in a tab-separated file named after the workload file and the TAG variable included in the control file. For example, after running:

```
$ leapfrog sshfs-disk-workload 5
```

We find that the output has been backed up in `/tmp/1291266494-sshfs-disk-workload/`. 1291266494 was the UNIX time when the test began; this ensures that no subsequent test will overwrite data from a previous test. More importantly, in `/tmp/1291266494-sshfs-disk-`

`workload/reports/`, we find three files, `copy.tsv`, `remove.tsv`, and `untar.tsv`. Each of these files contains aggregate data (one row per repetition) corresponding to a workload tag that was present in the control file. The reporting tool prints the data in a tab-separated format, in addition to the human-friendly format printed to the console. This conveniently and automatically separates data from multiple repetitions into a format which is ready to be imported into a tool such as `R`, Open Office Calc, `gnuplot`, or Excel for analysis.

For example, `copy.tsv` contains the following information:

```
# CPU (J) HDD (J) BR (J)  RAM (J) CPU (W) HDD (W) BR (W)  RAM (W)
8.669092	6.065411	1.179657	2.988264	2.665445	1.864903	0.362703	0.918787
8.002690	5.107588	1.225798	2.701306	2.671481	1.705030	0.409200	0.901758
7.937569	5.246586	1.133595	2.783591	2.659865	1.758121	0.379866	0.932776
8.083968	5.213154	1.228385	2.837290	2.669827	1.721706	0.405689	0.937049
7.245004	4.713862	1.074891	2.629322	2.665368	1.734185	0.395442	0.967303
```

4.5 Manually selecting tabular reporting

By default, `user_caliper_report.py` does not print tabular output. However, Leapscripts invokes `user_caliper_report.py` with options to enable this feature. You can manually select this style of reporting even if you are not using Leapscripts. For example, to write TSV files to `/tmp/reports` (in addition to the human-readable console output), execute:

```
$ user_caliper_report.py -t /tmp/reports sync-data.txt
```

4.6 Remote data processing

Data processing on the LEAP platform typically takes much longer than the actual experiments, and uses the full processing power of the LEAP platform. This reduces the number of experiments that can be tested, because running experiments while processing data would produce innacurate experimental results. Some users may wish to collect data on the LEAP platform, and transfer it to another system for processing.

Three files are needed to fully process the data. First, you need the raw data itself (`data.txt` in our examples). Then,for synchronization, you need the kernel message file (`/var/log/messages`), and the energy caliper control file (`/ramdisk/Energy_Caliper_Control_File`).

4.6.1 Remote synchronization

To perform the synchronization step on a remote server, users must provide the raw data and the location of the kernel log containing synchronization messages to the synchronization script. This is done by specifying the log file as the second argument to `sync.py`, like so:

```
$ sync.py data-copy.txt messages-copy.log > sync-data.txt
```

If no second argument is specified, the script assumes the kernel log is at `/var/log/messages`.

4.6.2 Remote reporting

Running the reporting tools on a remote server requires you to provide the control file containing the “energy caliper” time stamps for the experimental data. Specify this path with the -e switch, typically used with the -t switch described above. For example:

```
$ user_caliper_report.py -t /tmp/123123123-workload/reports -e /tmp/123123123-
workload/Energy_Caliper_Control_File sync-data.txt
```

This command manually specifies the synchronized data file and control file to use, and specifies an output directory for tabular data.

4.7 See also...

For more information on LEAPfrog, see the code in `/usr/atom_LEAP/code/frog/frogscripts.sh` is a set of functions for managing the processes, `leapfrog` is the launcher, and the `*-workload` files are sample workloads.

Please report bugs in the LEAPfrog tools to Peter A. H. Peterson <pahp@cs.ucla.edu>