# Group 2 Project– Final Report

# Data Movement and Power Measurements

# CS533 Spring 2018

## Portland State University

## June 9, 2018

Group Members:

* Alex Kelly <keap@cs.pdx.edu>
* Shikha Shah <shikha2@pdx.edu>
* Ajinkya Shinde <ajsh2@pdx.edu>
* Jason Graalum <jgraalum@pdx.edu>

GitHub Repo: <https://github.com/jasongraalum/CS533_Spring2018_Group2_Project>

Goals:

1. Write benchmark codes that target specific configurations expected to be power efficient or power inefficient.
2. Conduct a study using the benchmarks to actually measure the power consumption.

Background:

In a typical “pipeline” process, data is move from processor to processor where different operations are performed. The assumption is that there is an efficiency gain by allowing the processors(or cores) specialize – think automobile assembly line. However, there may be a flaw in this thinking. As each core is general purpose, the efficiency gains, if any, from specialization, may be small.

Given a large enough data set, it may be more efficient to move the operations, instead of the data, from core to core. Our work tests this idea.

Hypothesis:

Given a large enough data set, relative to the size of the local memory(local being relative itself), the power consumption will decrease if the operation is moved from between cores as compared to moving the data between cores.

# Methodology

There are two main challenges to evaluating the power usage with in a system. The first is finding a measurable metric which is directly related to the case to be evaluated. In our case, measuring the power due to data movement between the core, L1 cache, LLC and main memory must be done indirectly. This is because we do not have physical access to the busses or the ability to physically isolated them. The second challenge is due to the complex interaction between the core, CPU, computer hardware system and the operating system. Each of these impact the total power of the system and introduces “noise” in our power measurement. An attempt to account for and eliminate this noise is difficult. To address these two challenges, we made several methodology decisions.

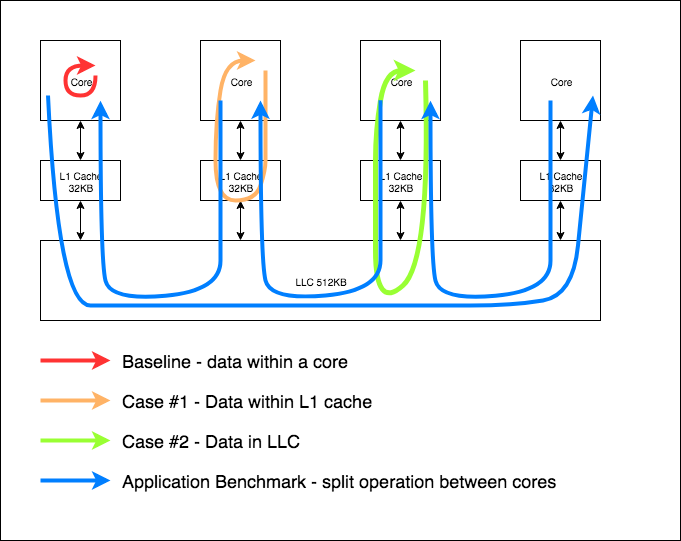
The first decision needing to be made was which hardware platform to use. We decided on the Raspberry Pi 3 mainly due to access. Also, using a system which we had complete control over allowed us to experiment with the operating system and job control. For instance, we were able to disable many of the background jobs running to help eliminate some of the noise in the power measurements. Also, we were able to hook up the power source to a power meter to directly measure the system power. Finally, the Raspberry Pi 3 has well defined(and small) cache sizes which helped limit the size of the overall testcases. The hardware and software specifications are listed below

#### Hardware Platform: Raspberry Pi 3[[1]](#footnote-1)

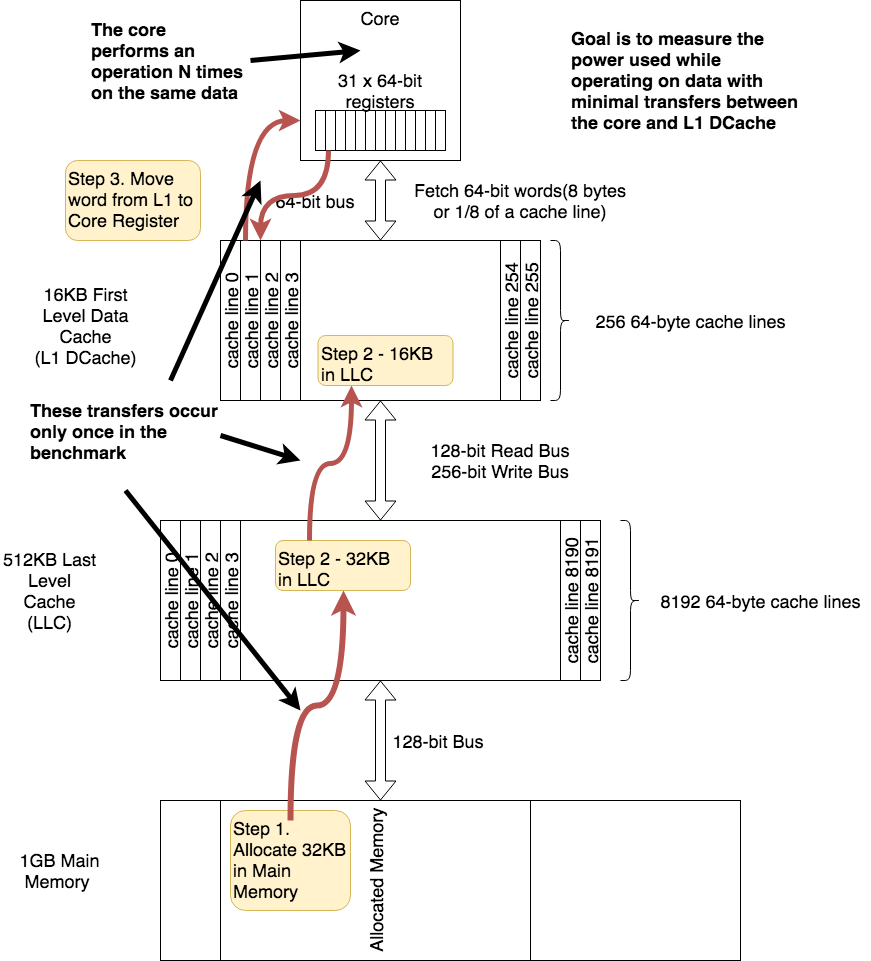
* SoC: Broadcom BCM2837
* CPU: 4 × ARM Cortex-A53, 1.2GHz
* GPU: Broadcom VideoCore IV
* RAM: 1GB LPDDR2 (900 MHz)

#### Software Configuration

* Kernel – Linux 4.14.34-v7+
* Customer built[[2]](#footnote-2),[[3]](#footnote-3)
* Add performance and power counter support
  + Need to capture all the details.



The second decision was which metrics to measure. An obvious choice was the total system power. We used a simple USB power meter as the Raspberry Pi 3 is powered through a standard Micro-USB port. The downside to using such a power meter is the relatively lower resolution. Combined with the fact that the Raspberry Pi 3 draws a low amount of quiescent power introduced even more noise into our measurements. Finally, we selected a set of system counter metrics to evaluate the operation of the cores. Using the Linux “perf” tool allowed us to record the number of instructions and the number of cache loads and misses per testcase. (In addition to “perf”, we also utilized the “schedtool” Linux tool. This tool allowed us to set the priority(niceness) of the running testcase to the highest level of 39. This decreased the chance that other jobs might interrupt a running test which would add additional noise to our measurements.



The final decision was which instructions to use to show significant differences in power consumption between the different data movements. We expected that there is increased power consumption due to loading data from the caches. Four test cases were decided on which would test four different data movements.

The first three test cases attempt to implement simple data movements between the core, the L1 cache and the Last Level Cache(LCC). We start with a baseline testcase(case #1) which attempts to limit the total data movement by operating on data currently resident in the core. The power measured during this test case should be limited to the quiescent power – i.e. idle power plus the power used in the operation. To be an effective baseline, all setup and operations must be consistent across all test cases. This means that we execute the same computation, which in our case is a simple addition. Also, the memory allocation must be consistent. This memory allocation is dictated by case 3 which requires the most amount of memory.

Figure : Test Case #1 Data Movements

The next two cases measure the power between the core and L1 cache and the core, L1 cache and LLC respectively.

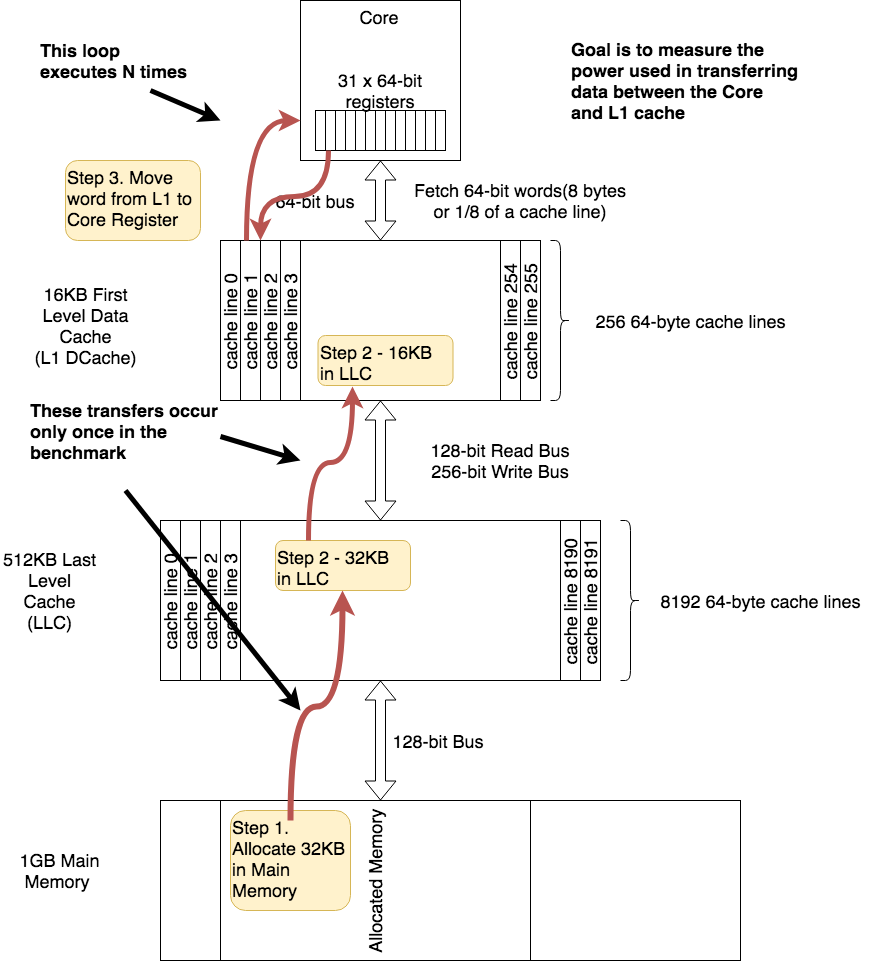
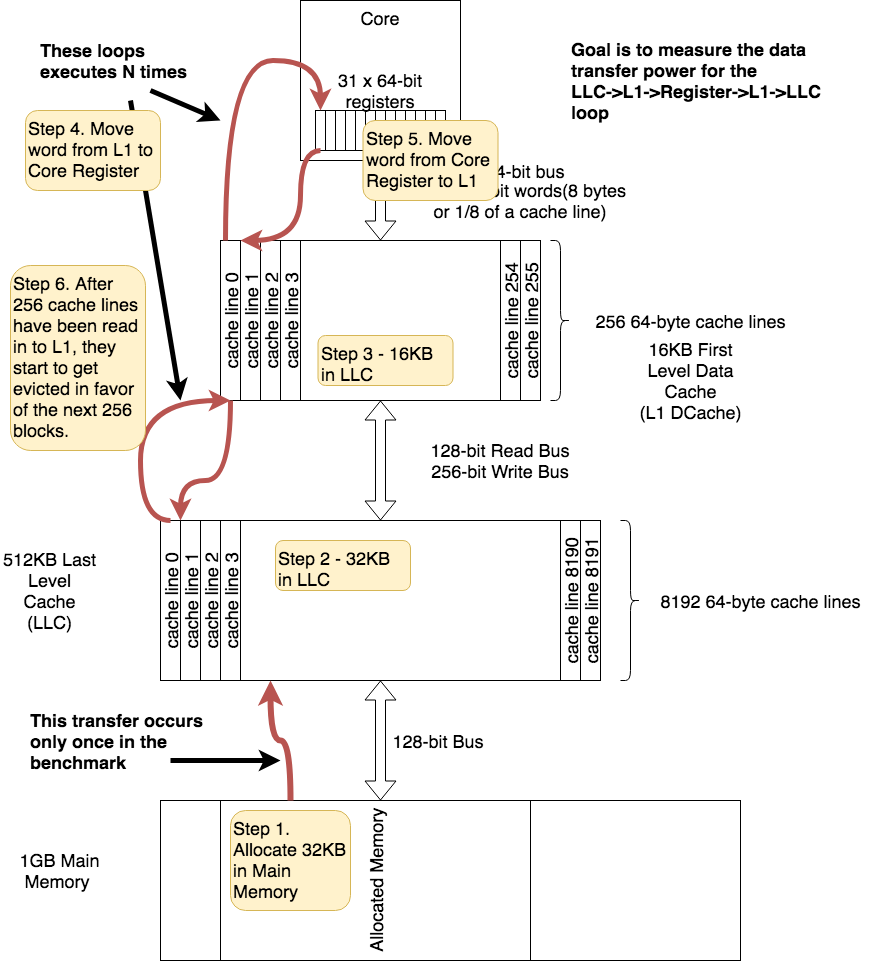


Figure : Test Case #2 Data Movement

To test the L1 cache power in case #2, we must force the loading of memory between the L1 cache and the core registers. To achieve this we load from a unique location for each loop of the test case. As shown in the diagram to the left, on each loop iteration, data is loaded into the core from alternating L1 cache lines. Because we limit the access from just two cache lines, we expect that there will be limited data moved between the L1 cache and LLC.



For case 3, we need to force a L1 cache miss so that a new cache block is loaded from the LLC on each iteration. To do this, we need to be sure to access a memory location from a different L1 cache block on each loop iteration. We accomplish this be allocating an array that is twice the size of the L1 cache. (This is the amount of memory allocated for all testcases so that we are working with the same baseline.)

We need twice the size of L1 cache to guarantee that every access causes an L1 cache miss. On the first half, each cache line is loaded from the LLC into the L1. Each increment the array offset by the size of a cache line to ensure that we are accessing a different cache line. On the second half of the iterations, each cache line is replaced and loaded with a new block. Therefore on every access, a new cache line should be loaded from the LLC.

Figure : Test Case #3 Data Movement

# Benchmarks

#### Data Movement Types

There are two types of data movement to be considered: Instructions and Data. The instructions must be fetched from, at a minimum, L1 Cache. This movement of data is unavoidable. If we assume that instructions for all benchmarks fit in L1 Cache, we can eliminate the corresponding power as it is equal across all benchmarks. All benchmarks should also start out with an equivalent amount of data in main memory and load it into L1 cache and LLC in the same way. Also, the operations on the data must be equivalent so that the power used by the core is the same across benchmarks.

#### Measurement Types

1. Baseline - With-in Core – only accessing Core Registers
2. Between Core and L1 Cache
   1. Data limit of 32KB
3. Between a single Core and LLC
4. Between Cores via LLC
   1. Core 1 L1 Cache -> LLC -> Core 2 L1 Cache
5. Between Cores via Main Memory
   1. Core 1 L1 Cache -> LLC -> Main Memory -> LLC -> Core 2 L1 Cache

#### Measurement Methodology

Goals

1. Get power cost of moving a single value between the core and various levels of memory.
2. Get power cost of moving a single value from core to core.

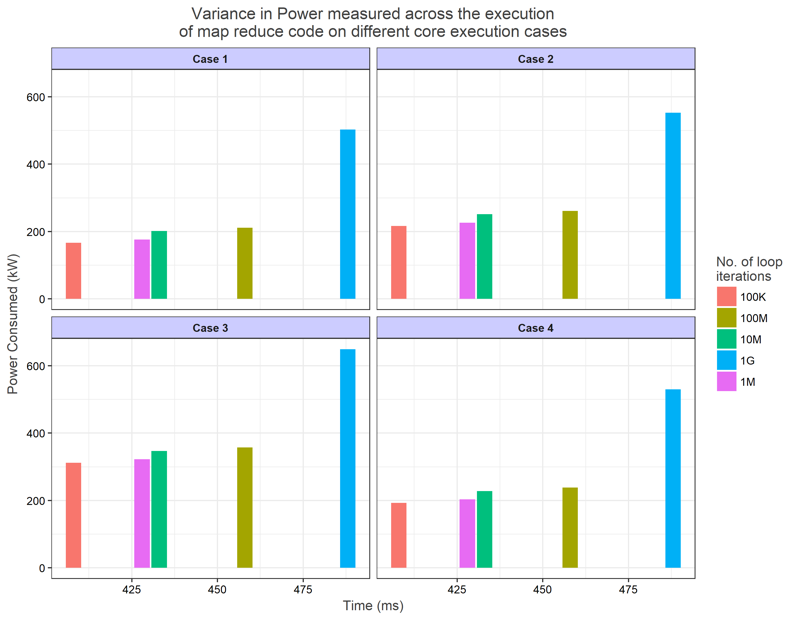
#### General Steps

1. Move instructions into iCache
2. Move data into Core <- source depends on benchmark goal
3. Operate on data
4. Move data out of Core <- destination depends on benchmark goal

#### Cases

1. Case #1 - With-in Core <- this is the baseline power
   1. Load instructions from main memory into L1 cache
   2. Move data into LCC and L1 cache from main memory
   3. Move data into core register
   4. Operate on in-register data – avoid moving data out of Core registers
      1. Perform N operations
   5. Output data to Main Memory
   6. Read perf counters
2. Case #2 - Between Core and L1 cache
   1. Load instructions from main memory into L1 cache
   2. Load data into LCC and L1 cache from main memory
   3. Load data into core register
      1. Accessing the same cache line is OK as long as the data is moved from L1 to a core register
   4. Operate on data
   5. Store data in L1
      1. Be careful not to access LLC
   6. Repeat last three steps N times
   7. Output data to Main Memory
   8. Read perf counters
3. Case #3 - Between Core and LLC
   1. Load instructions from main memory into L1 cache
   2. Load data into LCC and L1 cache from main memory
   3. Load data into core register
      1. Each access must cause a load from LLC – there are 256 cache lines in the data cache of each core – each of 64-bytes
   4. Operate on data
   5. Store data in L1 and LCC
      1. Must be sure to push data to LCC
   6. Repeat last three steps N times
   7. Output data to Main Memory
   8. Read perf counters
4. Application Benchmark – Core to Core Data Movement
   1. Baseline
      1. Move all data into L1 cache and LCC from main memory
         1. Data set should be large enough that multiple cache misses will occur
      2. Execute 4 operations on all the data on a single core.
      3. Store data in main memory from single core L1
   2. Split Data
      1. Move ¼ of the data from main memory into L1 cache of each core
      2. Execute 4 operations on all the data in each core
         1. Large data – multiple fills of L1 cache
      3. Store data in main memory from each core L1 cache
   3. Core to Core
      1. Move ¼ of the data into each L1 cache of each core
      2. Execute 1 operation in each core
      3. Store data back in main memory
      4. Shift data to each core and repeat for each of the 3 remaining operations

Results – The results below are not real data. This is a mock-up of our expected data display. We are in the process of taking data.



1. https://www.raspberrypi.org/magpi/raspberry-pi-3-specs-benchmarks/ [↑](#footnote-ref-1)
2. https://www.raspberrypi.org/documentation/linux/kernel/building.md [↑](#footnote-ref-2)
3. https://www.raspberrypi.org/documentation/linux/kernel/configuring.md [↑](#footnote-ref-3)