Modeling Thermal Effects of Hardware Events Based on Performance Monitoring Unit Event Counters

Eric Stevens

https://github.com/Eric-D-Stevens/thermal_aware_process_scheduling

1 Introduction

According to Arrhenius equation, an increase of temperature by 20 degrees Celsius results in a reliability decrease of an electronic device by 50 percent [1]. Modern CPUs protect themselves against overheating through techniques like frequency scaling and DVFS [1]. While effective at reducing temperature, these techniques diminish performance by slowing processing down.

This project explores the viability of utilizing on chip performance monitoring units (PMUs) to model CPU heat generation. If PMU metrics can provide insight into the relative heat generation of processes, a scheduling algorithm can be devised that allows processes that generate less heat to be scheduled more often, reducing short term temperature spikes that harm the CPU and/or result in performance degrading safety mechanisms to kick in.

In this project, a method for collecting PMU hardware counter data and CPU temperature data is discussed and implemented. PMU events are counted for a series of random programs that attempt to exercise a range of CPU operations that the PMU can monitor. Simple machine learning algorithms are applied to the collected PMU data in an effort to model the change in temperature of the CPU. The capabilities of the resulting model are discussed; its strengths and weaknesses are evaluated.

The goal of this project is to show that thermal characteristics of CPUs can be modeled through the use of on board PMU, setting the stage for a thermal aware process scheduler that can reduce temperature spikes on fully loaded processors.

2 Tools

This project relies on several open source tools that are available for most modern Linux distributions and CPU architectures that run Linux. The hardware used for this project was a Raspberry Pi 3 running Debian 9 (Stretch), but the code should work on most Linux systems. The information here is provided to introduce the tools used as well as to help the reader understand what was done in the project and how to run the code.

2.1 Linux `perf` - PMU event monitoring [2]

The Linux `perf` utility, also known as `perf_events` [2], can run on most Linux distributions and allows for user-level interaction with the performance monitoring unit. `perf` is most commonly used as a command line utility, allowing for quick profiling and reporting on software execution characteristics. This project uses the `perf_event.h` [3][4] library directly in an effort to achieve more control over how PMU data is collected. It allows users to specify which PMU events are to be monitored, how and when to begin collecting event data, and read/reset the PMU counters.

2.2 `stress` - Exercising the CPU [5]

In order to model the thermal effects of PMU events, the CPU must be exercised and the event counts and resulting temperature changes recorded. Initially, an effort was made to write custom programs that exercise specific counters. This process became arduous and was abandoned for the sake of time.

`stress` is an open source Linux tool designed to impose a load on a system [5]. There are a limited number of operations that you can run with `stress`, but by combining these tasks in a variety of ways we can get a variety of benchmarks. The `stress` tool was used to exercise the CPU for all data collected.

2.3 Python - Scripting and Modeling

While it was necessary to work in C for directly gathering information from the PMU, Python scripting was used for all other tasks. These tasks include both the automation of running the C program that collects PMU data and performing the machine learning tasks to generate the model.

A Python script was written to manage running the PMU data collection program. This script plays a role in both generating random programs to collect data from and monitoring CPU temperature to ensure data collection runs are executed over a similar temperature range.

Pythons Pandas library [7] was used for data collection and manipulation and Pythons Scikit-Learn library [6] was used for machine learning tasks. These tasks include data preprocessing, separating data into training and validation sets, fitting models, and leveraging trained models to create visualizations. Visualizations were created with Pythons Matplotlib [8].

3. Implementation

The implementation of this project can be divided up into two distinct sections: data collection and model building. C code was written to interact directly with the hardware in order to exercise the CPU while collecting information from the PMU and Python was used to automate running the C code. Python tools are used for analyzing the collected data and building a model. Before diving deeply into either one of these sections, the hardware and its PMU will be described.

3.1 Hardware

The hardware being characterized sits on a Raspberry Pi 3 with a Broadcom BCM 2837 SoC with an ARM Cortex A53 processor [9]. The ARM Cortex A53 processor is equipped with 6 configurable PMU counters, as well as a dedicated counter for CPU cycles [10]. This means that at any given time six different events can be captured simultaneously.

3.2 Data generation and Collection

As mentioned in the tools section, the `stress` utility is used to generate a load on the CPU. Code written in C leverages the `perf_events.h` interface to monitor the PMUs while these loads are being executed. Python automates the process of collecting data over a large number of runs. The combination of these three techniques represents the data generation and collection process. This section will cover each of them in greater detail.

3.2.1 `stress`

`stress` is not a benchmarking tool but is rather a tool designed to enable developers to put a subsystem under a specific load [5]. An example of when this might be useful is when a kernel or lib C programmer wants to evaluate the possibility of denial-of-service attacks. It is not the perfect tool for the purpose of this project as it can generate i/o bound workloads. However, it provides an expedient and consistent way to test a variety of differing loads. **Table 1** provides a description of the different options `stress` can be called with and what they do [11].

Option	Description
-t, --timeout N	Timeout after N seconds
-c, --cpu N	Spawn N workings spinning on sqrt()
-i, --io N	Spawn N workings spinning on sync()
-m, --vm N	Spawn N workings spinning on malloc()/free()
-d, --hdd N	Spawn N workings spinning on write()/unlink()

Table 1: `stress` utility option for imposing a workload on a system

In the experiment, timeout is set to 120 seconds. The number of workers is randomly selected for each of the options in a range from 0 to 3. All of the workers are forced to run on the same, single CPU, which is monitored by the PMU, as discussed in the next section.

3.2.2 `perf_events.h`

The implementation of the program to monitor and record PMU events was the most intricate and sensitive part of this project. Many time-consuming mistakes were made. While the use of the command line utility, `perf`, is fairly well documented [12], it was difficult to find documentation regarding the proper use of the underlying C interface. A 10-page paper could be written about the mistakes I made using this library alone, so there may be some information missing in this section.

All of the project code for collecting PMU events is in a single file, `single_cpu.c` in the main repository directory. This section describes the `perf_events.h` interface and how it is used in `single_cpu.c`.

Events Used

`perf` is not always aware of what PMU capabilities a given CPU architecture supports. There is a process of elimination that needs to occur to determine which events can be monitored. There is a set of predefined events that most Linux distributions assume can be monitored. As mentioned in the hardware section, the ARM Cortex A53 has 6 PMU counters, meaning that six different events can be monitored simultaneously. If one wishes to monitor more events than just the six, then multiplexing needs to occur, swapping in and out events on counters and aggregating the results in a meaningful way. Due to a false assumption that the automatic multiplexing that occurs with the command line `perf` utility would also occur when using `perf_event.h`, multiplexing was not performed in this experiment. Therefore only six events are monitored. The names and descriptions of the monitored events can be seen in **Table 2**.

Event Name	**Description**
PERF_COUNT_HW_CPU_CYCLES	Total cycles. Be wary of what happens during CPU frequency scaling.
PERF_COUNT_HW_INSTRUCTIONS	Retired instructions. Be careful, these can be affected by various issues, most notably hardware interrupt counts
PERF_COUNT_HW_CACHE_MISSES	Cache misses. Usually this indicates Last Level Cache misses
PERF_COUNT_HW_BRANCH_MISSES	Mispredicted branch instructions.
PERF_COUNT_HW_BRANCH_INSTRUCTIONS	Retired branch instructions.
PERF_COUNT_HW_CACHE_LL_read_miss	Measuring Last-Level Cache for read misses
PERF_COUNT_HW_CACHE_L1D_read_miss	Measuring Level 1 Data Cache for read misses

Table 2: The 7 PMU events monitored for the experiment

This set of events represents a small fraction of the events that could be monitored. A more complete version of this experiment would perform multiplexing and gather many more events.

Opening Up Events for Monitoring [4]

A perf_event_attr is a C structure, defined in perf_event.h, that abstracts the concept of a PMU event. PMU events can be scheduled individually or in groups. Setting up PMU events as groups ensures that the action taken on the PMU (reading/activating/resetting) occurs simultaneously. The function perf_event_open is implemented in order to establish a PMU event to be monitored on the PMU. This function wraps a system call to __NR_perf_event_open. This function can be seen in **Figure 1**.

```
static long
perf_event_open(struct perf_event_attr *hw_event, pid_t pid,
                    int cpu, int group_fd, unsigned long flags){
    int ret;
    ret = syscall(__NR_perf_event_open, hw_event,
                    pid, cpu, group_fd, flags);
    return ret;
}
```

Figure 1: The perf_event_open function establishes the PMU event on the PMU [4]

The first attribute of the perf_event_open function is the perf_event_attr structure itself. perf_event_attr will be discussed below. The next two attributes are pid and cpu, which are the PID of the process to monitor and the CPU on which to monitor it respectively. There are options for monitoring all processes on a given CPU or monitoring a process on "any" CPU. However, the meaning of "any" led to confusion in the project, eventually leading to the decision to only run the stress workloads on a single CPU. This ensures that events are being properly monitored. The next parameter, group_fd, is the file descriptor of the "group leader". The group leader is the first event declared in a given group, created by setting group_fd to -1. Following the declaration of the group leader, subsequent events hand in the group leaders file descriptor to establish themselves as a member of the group.

Setting Up Events [4]

Each event to be monitored must have its own perf_event_attr structure and must be opened with the perf_event_open function. These perf_event_attr structures have many fields that determine the way they interact with the PMU. **Figure 2** shows a loop in which perf events structures are instantiated and opened.

The config field is set to the macro representing the event as seen in **Table 2**. Setting the disable field initializes the structure as inactive on the PMU. This is only done to the group leader. We don't set the disabled field on the rest of the group members, because they follow the leader and will remain inactive until the leader is activated. The exclude_kernel and exclued_hv fields prevent the PMU from monitoring kernel and hypervisor events respectively. The most important field for this experiment is inherit. Setting this field ensures that all events occurring in child processes will inherit the monitor and be counted. This is crucial because the stress utility workers are launched as child processes. After the per_event_attr struct has been instantiated, it is handed to the perf_event_open function. If it is the first to be inserted it will be passed a group_fd of -1, declaring itself as the group leader. Otherwise, the previously declared group leaders file descriptor will be passed in as the group_fd, adding the event to the group. The group can have up to 6 members since there are 6 configurable PMU counters in the ARM Cortex A53.

```
for(int i=0; i<num_hw_events; i++)
{
    memset(&pat_arr[i], 0, sizeof(struct perf_event_attr));
    pat_arr[i].type = PERF_TYPE_HARDWARE;
    pat_arr[i].size = sizeof(struct perf_event_attr);
    pat_arr[i].config = pe_hw[i];
    if(i==0){pat_arr[i].disabled = 1;}
    else{pat_arr[i].disabled = 0;}
    pat_arr[i].exclude_kernel = 1;
    pat_arr[i].exclude_hv = 1;
    pat_arr[i].inherit = 1;

    if(i==0){fd_arr[i] = perf_event_open(&pat_arr[i],0,0,-1,0);}
    else{fd_arr[i] = perf_event_open(&pat_arr[i],0,0,fd_arr[0],0);}
    if (fd_arr[i] == -1){
        fprintf(stderr, "Error opening leader %llx\n",
                                            pat_arr[i].config);
        exit(EXIT_FAILURE);
    }
    printf("FD%d: %d \t ITEM: %s\n",i,fd_arr[i], hw_name_arr[i]);
}
```

Figure 2: Loop for instantiating PMU events and establishing them with the PMU

Recording a run

After all of the events have been established, the events can be activated and the stress workload can be executed. The stress workloads are called with a timeout value of 120 seconds. Events are read from the PMU once approximately every one second. It is not always exactly 1 seconds because occasionally one of the processes will block the reading of the PMUs.

Along with the event counter values, the temperature of the CPU and the actual time the counters were accumulating are stored. they are appended to a CSV file. After the stress process times-out, it shuts down, triggering a SIGCHLD in the main process that results in termination. At this point the run is complete and the data for that workload has been collected.

3.2.3 Python Data Generation Automation

The methods mentioned above are intended to collect information for a single run. `stress` parameters are passed in as command-line arguments to the executable. The Python script manages the generation of the data set by randomly selecting `stress` options and then passing them as parameters to the data collection executable.

Another important function of the Python script is to monitor the temperature of the CPU in between runs and wait until it drops below a certain threshold prior to starting the next run. After executing each data collection run, the Python script sleeps in 5 Second intervals and reads the temperature of the CPU. Once the temperature has dropped back to a baseline, it launches the next run. This process is repeated until the desired number of runs have been executed. In this way we collect information over the same temperature ranges for each run.

3.3 Modeling [6][13]

The model is designed to predict the **change in temperature per second** given a current temperature and a set of PMU counter values. Predicting change in temperature instead of total temperature reduces the range of likely values. In this section the data preprocessing and modeling steps to predict change in temperature are discussed.

3.3.1 Preprocessing

Data preprocessing was a significant challenge in this project. The main challenge for modeling was the granularity of the CPU thermometer measurements. This led to a situation where when a process that was not generating much heat was running, a CPU could sit at the exact same temperature for many PMU read cycles, biasing the data towards a zero temperature change. In order to combat this issue, windows of time were sampled and then aggregated by dividing all of the counters and the temperature changes by the amount of time the counter was active. All data files in the training set we're sampled 100 times with window sizes ranging from 4 to 16 seconds. This collection of aggregated windows became the training data. Prior to training or prediction, data is scaled feature-wise to between 0 and 1, preventing the relative magnitude of the counter values from having greater influence than any other. [14]

3.3.2 Model Selection

The model that was selected was a simple linear model with ridge regularization. [13] This model was selected for its simplicity and for how easy it is to interpret the results. Ridge regression has a single regularization parameter (alpha in sci-kit learn) that biases the model in exchange for reduced variance. In other words, it forces the model to be less sensitive to input variations than training a least squares model would. The alpha hyperparameter was set to a value of 1 for this experiment.

The use of polynomial features and more powerful models like support vector machines were explored, but due to difficulties during the project and a shortage of time they were not examined with enough depth to present results on them in this paper.

4 Results

The results of the experiment indicate that the method of using PMU counters to predict CPU temperature has merit, however the experiment is incomplete, and requires more development in order to build a useful model that could be used in a thermal aware scheduler. **Figure 3** shows the distribution of training data event counter values after preprocessing. These distributions represent the range and likelihood of combinations of events in the training data. It is obvious from the shapes of these distributions that there is a significant correlation between many of the events. This can have a negative impact on the linear model.

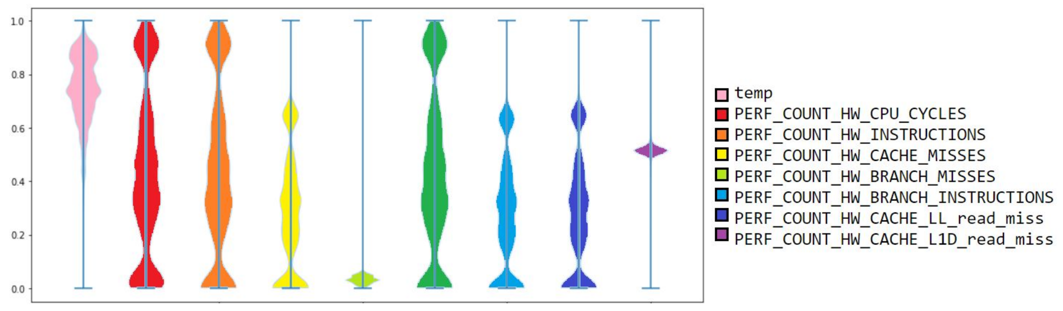

Figure 3: Realive distribution of PMU events in training data

PMU Event	**Linear Coefficient**
temp	-0.219932
PERF_COUNT_HW_CPU_CYCLES	0.074059
PERF_COUNT_HW_INSTRUCTIONS	0.034690
PERF_COUNT_HW_CACHE_MISSES	0.166357
PERF_COUNT_HW_BRANCH_MISSES	0.622627
PERF_COUNT_HW_BRANCH_INSTRUCTIONS	0.053337
PERF_COUNT_HW_CACHE_LL_read_miss	-0.029165
PERF_COUNT_HW_CACHE_L1D_read_miss	0.166737

Table 3: Linear coefficients of trained ridge regression model

Table 3 shows the linear coefficients resulting from training the ridge regression model. The temp coefficient is negative, which is to be expected. A negative temperature coefficient indicates that the higher the temperature of the CPU is, the higher the values of counters with positive coefficients must be to overcome the negative temperature coefficient and result in a positive (increasing temperature) output.

The last level cache read parameter is also slightly negative, but this may be a misnomer due to the correlation with cache misses (PERF_COUNT_HW_CACHE_MISSES). One thing that was noticeable in most variations of the experiment was that Branch instruction misses have a high correlation with temperature increase. Why this is the case is an issue to be explored.

It is difficult to determine how meaningful any of the coefficients actually are. Regardless, the goal of the model is to predict temperature change given a set of inputs. **Figure 4** shows actual runs that the model has never seen before. The red lines show the true temperature reads of the CPU over time. To generate the green lines, the first true temperature reading from the run is provided as the temperature parameter. From there the model is fed the counter values and makes a prediction for the change in temperature. That prediction is added to the previous temperature reading in order to create the predicted temperature and temperature parameter to be fed into the model for the next time point. This is done iteratively through the remainder of the data. In other words, the model only sees the counter values at each time step and the very first true temperature value to generate the green line.

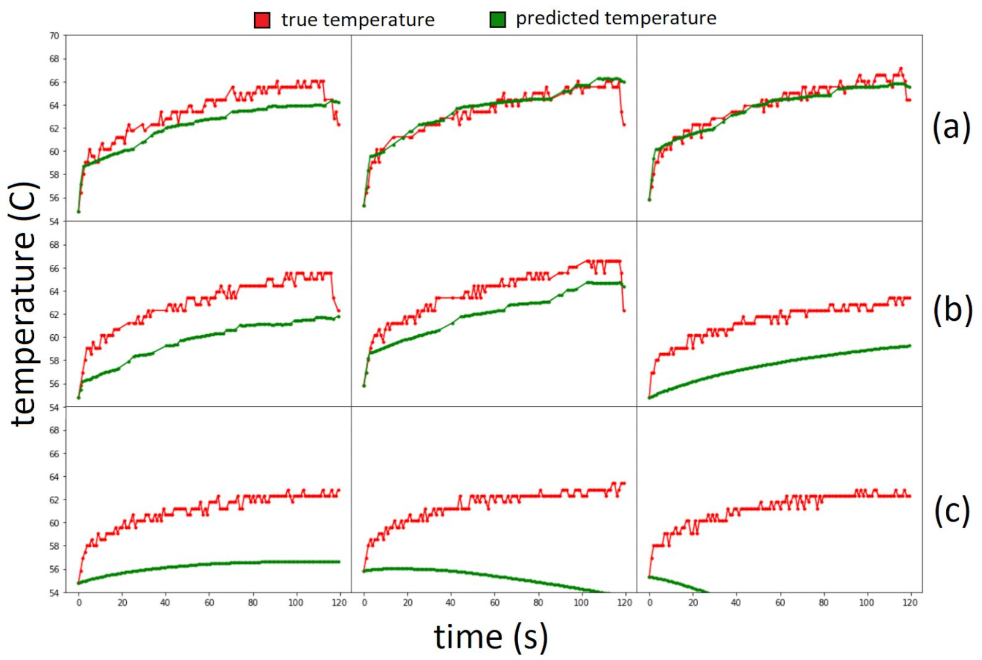

Figure 4: True (red) and predicted (green) temperature over the course of runs that have not been seen by the model. (a) Best performance runs. (b) Average runs. (c) Worst case runs.

We can see that performance of the model varies wildly. Most cases look roughly like Figure 4 (b) where the prediction line underestimates the temperature, but follows the general trend of the true temperature line. If all predictions looked like Figure 4 (a) we would have a great model that might be worth moving to the scheduler building phase with. Predictions like those in Figure 4 (c) are utterly useless, and could be dangerous if used in a scheduler. So what is going on here? There are many shortcomings in the project that will be discussed in the following section.

5 Shortcomings (Room for Improvement)

This project can be seen as a rapid, end-to-end prototype of a process to gather PMU data and model CPU temperature changes.Certain results imply that the general concept is sound. There is so much room for improvement, and it is yet to be seen how well temperature changes can be modeled from PMU counters.

Workload Generation

Using stress was probably not the right choice for this project. Stress generates workloads that are not necessarily fully loading the CPU. I/O bound processes may be a factor leading to some of the worst predictive results in the experiment. Finding another tool that is meant for varying CPU bound workloads would enable the generation of datasets that are more relevant to the problem being addressed in the project.

Event Monitoring

The mistake of assuming that using the perf_event.h library would automatically multiplex events was a major setback. Capturing only 6 events leaves a huge amount of potentially relevant data left unexplored. It is crucial for the success of the project that more events be monitored. This can either be accomplished by manually writing an event group multiplexer, or by learning more about tools that already do this (PAPI, perf). These tools were originally abandoned for the sake of simplicity in the hopes that I would have a more thorough understanding of the data I was gathering. They may be worth another look now that I have a better understanding of the mechanics of the project.

Modeling

The previous two shortcomings result in a situation in which it is not yet worth exploring more powerful models, in my opinion. Linear models are extremely simple, especially when only using the inputs in their raw form (no polynomial features) as was done in this experiment. The poorest performing test cases indicate that there is important data missing from the training set that can only be obtained by adding event multiplexing. There are many powerful modeling techniques that may perform better than ridge regression. Once there is a more complete dataset, exploration of these models could be fruitful.

6 Conclusion

While this project falls far short of a complete solution, I do believe that it demonstrates promise for a technique like this. Using machine learning to model CPU temperature changes on PMU events allows for a machine specific / workload specific prediction, an interesting outcome that I think is well worth continuing to explore.

6 References

[1] L. Dong, Hu, -Ch. Chang, H.K. Pyla, K.W. Cameron
System-level, thermal-aware, fully-loaded process scheduling
IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008 (April 2008)

[2] “perf_events Frequently Asked Questions.” perf_events FAQ, web.eece.maine.edu/~vweaver/projects/perf_events/faq.html.

[3] Torvalds. “perf_event.h.” GitHub, github.com/torvalds/linux/blob/master/include/linux/perf_event.h.

[4] “PERF_EVENT_OPEN.” Manpage of PERF_EVENT_OPEN, web.eece.maine.edu/~vweaver/projects/perf_events/perf_event_open.html.

[5] “Stress(1) - Linux Man Page.” Stress(1): Impose Load on/Stress Test Systems - Linux Man Page, linux.die.net/man/1/stress.

[6] “Scikit-Learn.” Scikit, scikit-learn.org/stable/.

[7] Pandas, pandas.pydata.org/.

[8] “Visualization with Python.” Matplotlib, matplotlib.org/.

[9] “BCM2837.” BCM2837 - Raspberry Pi Documentation, Raspberry Pi, www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2837/README.md.

[10] Arm Ltd. “Arm Cortex-A53 MPCore Processor Technical Reference Manual: About the PMU.” Arm Developer, ARM, developer.arm.com/docs/ddi0500/j/performance-monitor-unit/about-the-pmu.

[11] Gayan. “Stress Test Your Ubuntu Computer with 'Stress'.” Hectic Geek, 17 Dec. 2017, www.hecticgeek.com/stress-test-your-ubuntu-computer-with-stress/.

[12] “Tutorial.” Tutorial - Perf Wiki, perf.wiki.kernel.org/index.php/Tutorial.

[13] “Sklearn.linear_model.Ridge” Scikit, scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html.

[14] “Sklearn.preprocessing.MinMaxScaler” Scikit, scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html.