

Program Classification Using the PMU

Project Author: Eric Stevens, MS Candidate OHSU

Project Mentor: Bruce Irvin, PhD PSU

Motivation

- **Personal: Multidisciplinary project that exercises skills valuable for my career**
 - Computer architecture - PMU monitors microarchitectural events
 - Operating systems - User space tools and performance event configurations
 - Machine learning - Modeling outcomes based on PMU observations
- **Dataset generation**
 - Machine learning problem where the user gets to create the dataset instead of relying on existing datasets.
 - Results in the need to tune data collection hyperparameters as well as model hyperparameters.
- **Unique Problem**
 - Underutilization of a potentially valuable methodology
 - sparse documentation of tools
 - Not a substantial area of research
 - Dimensionality reduction when all features cannot be monitored simultaneously

Performance Monitoring Unit (PMU)

What is the PMU?

Extra hardware that monitors microarchitectural “events” and increments performance monitoring counters (PMCs) when they occur.

What are events?

The set of occurrences that PMU architects have enabled PMCs to count.

Event Examples

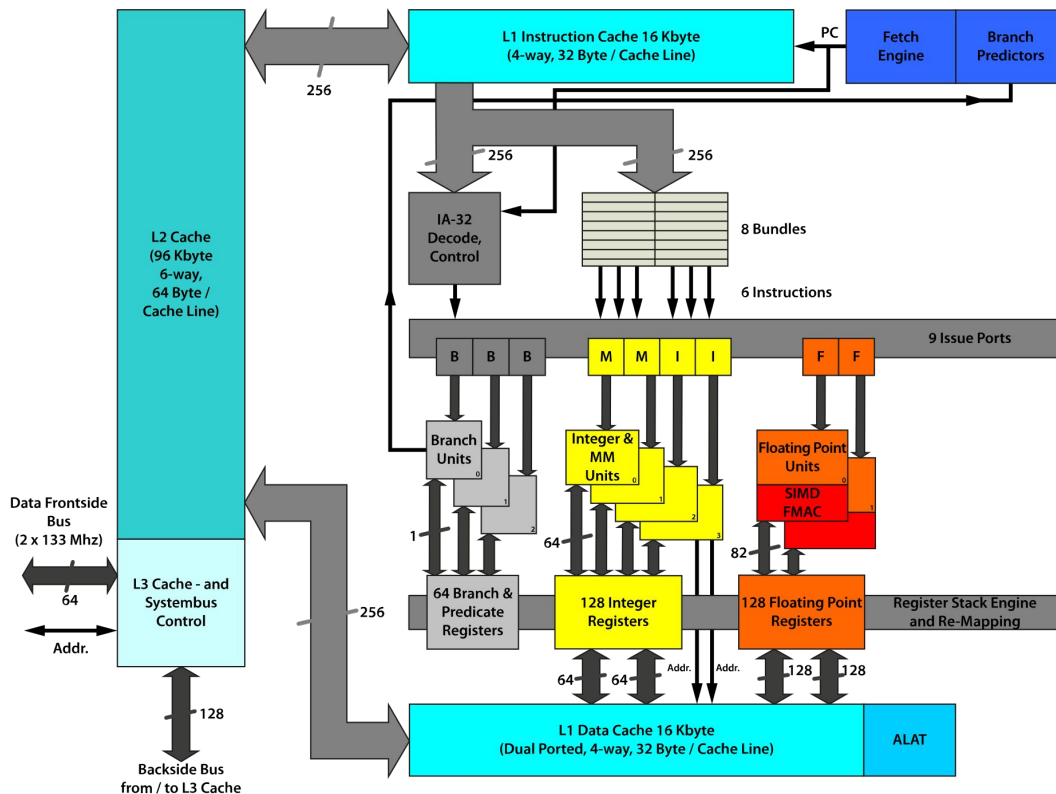
- Core clock cycles
- Instructions retired
- L1 Cache hits
- L2 Cache misses
- Branching instructions
- Branch mispredictions

PMU is Hardware

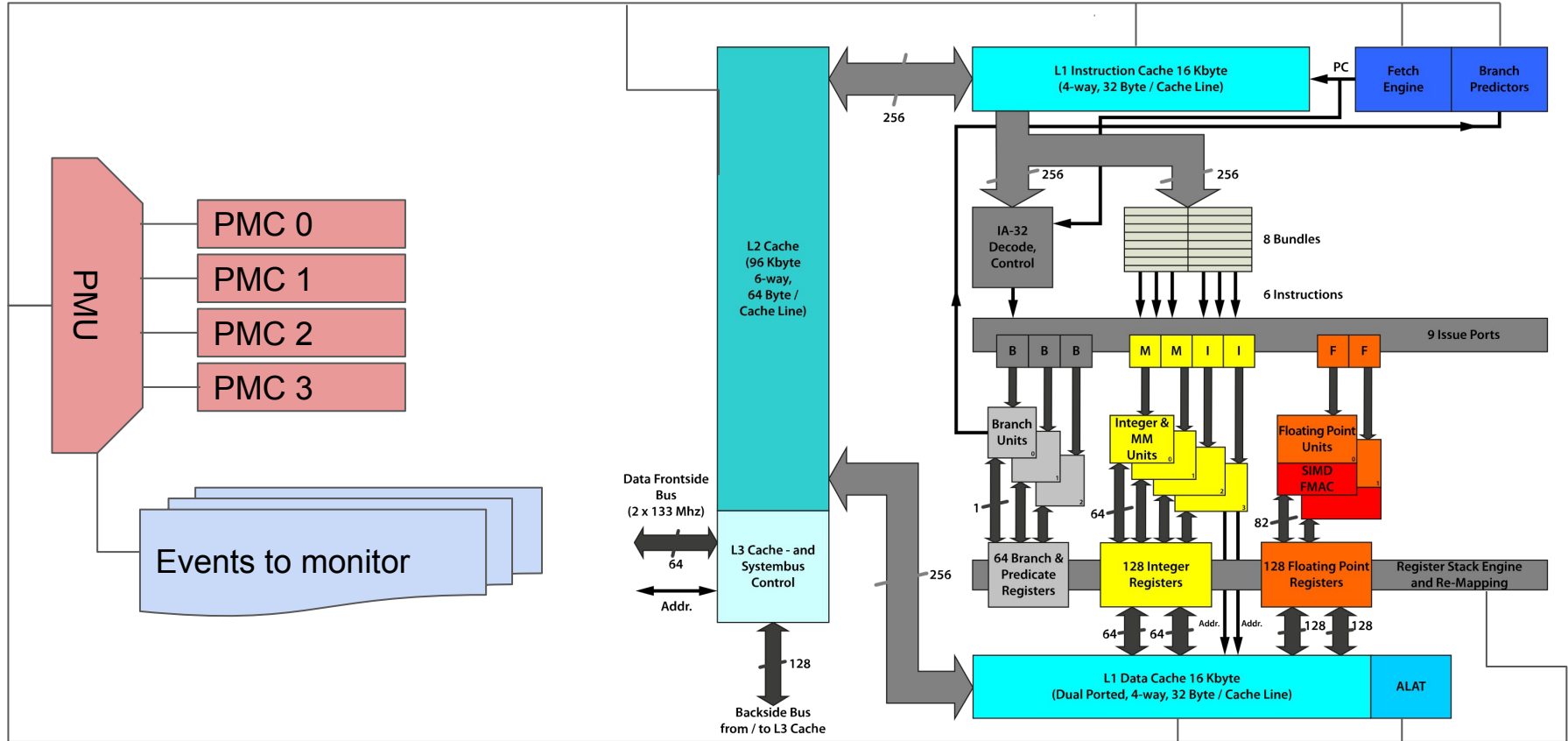
CPU diagrams have a lot of recognizable components

- Floating point units
- Caches
- Branch prediction
- Memory Busses
- etc...

You almost never see PMU hardware in a CPU diagram



PMU is Hardware



PMU Interface

The Intel IA-32 Software Developer's Manual is the end all be all for Intel CPU programming.

Referenced often when it comes to understanding Intel PMUs.



**Intel® 64 and IA-32 Architectures
Software Developer's Manual**

**Volume 3B:
System Programming Guide, Part 2**

NOTE: The Intel® 64 and IA-32 Architectures Software Developer's Manual consists of nine volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-M*, Order Number 253666; *Instruction Set Reference N-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831. Refer to all nine volumes when evaluating your design needs.

Order Number: 253669-060US
September 2016

PMU Chapters

CHAPTER 18 PERFORMANCE MONITORING

Intel 64 and IA-32 architectures provide facilities for monitoring performance via a PMU (Performance Monitoring Unit).

CHAPTER 19 PERFORMANCE-MONITORING EVENTS

This chapter lists the performance-monitoring events that can be monitored with the Intel 64 or IA-32 processors. The ability to monitor performance events and the events that can be monitored in these processors are mostly model-specific, except for architectural performance events, described in Section 19.1.

Non-architectural performance events (i.e. model-specific events) are listed for each generation of microarchitecture:

PMU Performance Event Select MSR

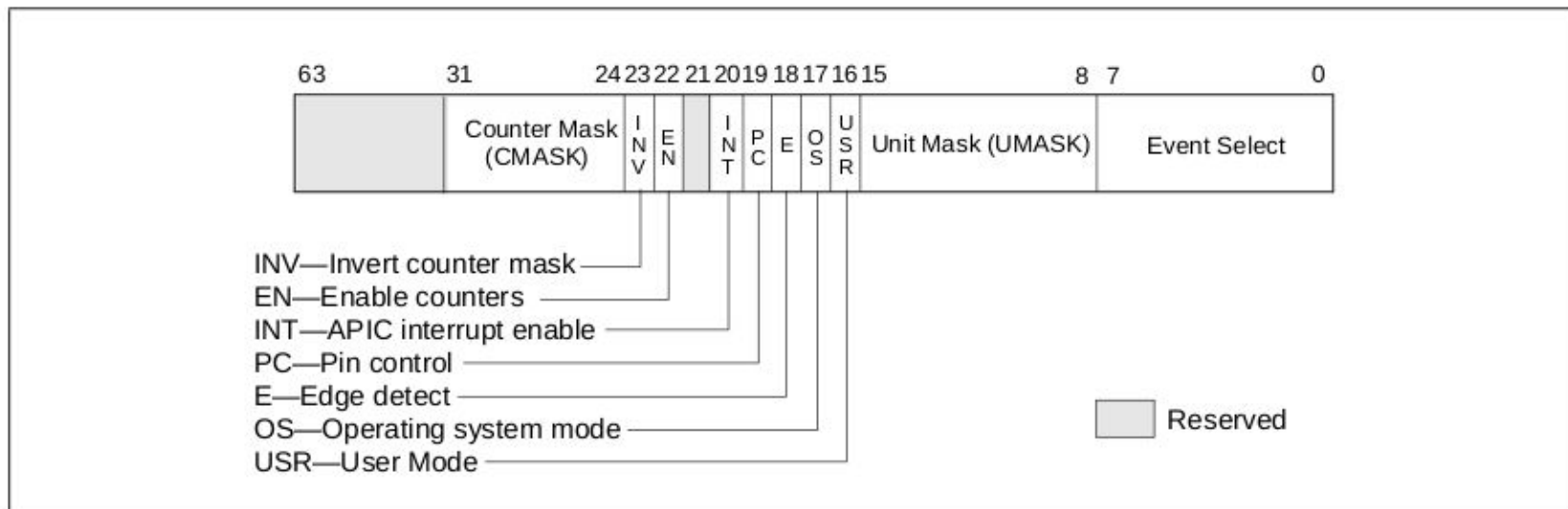


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

Event and Mask Example

Table 19-1. Architectural Performance Events

Event Num.	Event Mask Name	Umask Value	Description
3CH	UnHalted Core Cycles	00H	Counts core clock cycles whenever the logical processor is in C0 state (not halted). The frequency of this event varies with state transitions in the core.
3CH	UnHalted Reference Cycles ¹	01H	Counts at a fixed frequency whenever the logical processor is in C0 state (not halted).
C0H	Instructions Retired	00H	Counts when the last uop of an instruction retires.
2EH	LLC Reference	4FH	Accesses to the LLC, in which the data is present (hit) or not present (miss).
2EH	LLC Misses	41H	Accesses to the LLC in which the data is not present (miss).
C4H	Branch Instruction Retired	00H	Counts when the last uop of a branch instruction retires.
C5H	Branch Misses Retired	00H	Counts when the last uop of a branch instruction retires which corrected misprediction of the branch prediction hardware at execution time.

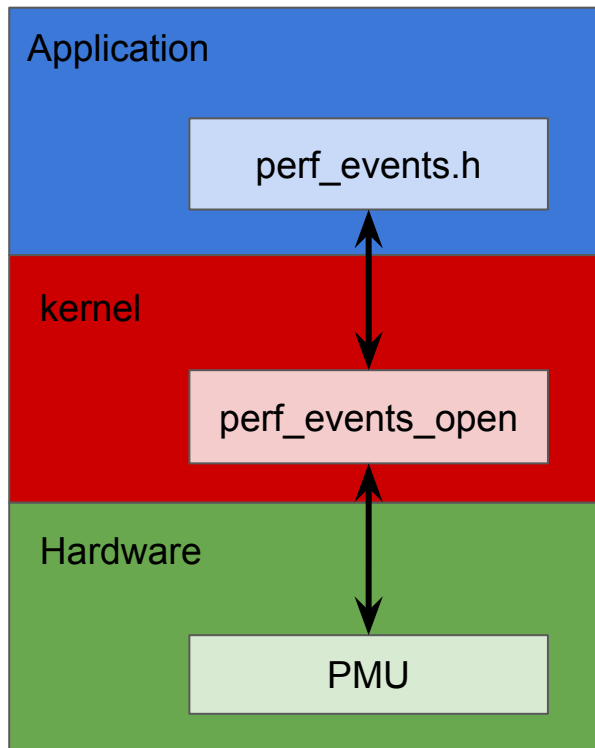
Interacting With the PMU

The PMU is part of the microarchitecture

One option is to write x86 assembly to control the PMU directly (lots of work)

To avoid assembly the kernel must expose an API to interact with the PMU

Perf_events is the Linux API and subsystem that allows for interaction with the PMU from C applications



perf_event_open

```
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;
}
```

pid == 0 and cpu == -1	This measures the calling process/thread on any CPU.
pid == 0 and cpu >= 0	This measures the calling process/thread only when running on the specified CPU.
pid > 0 and cpu == -1	This measures the specified process/thread on any CPU.
pid > 0 and cpu >= 0	This measures the specified process/thread only when running on the specified CPU.
pid == -1 and cpu >= 0	This measures all processes/threads on the specified CPU. This requires CAP_SYS_ADMIN capability or a <code>/proc/sys/kernel/perf_event_paranoid</code> value of less than 1.

perf_events_attr

- type - Type of PMU event
- inherit - If set, child processes events will be counted

```

struct perf_event_attr {
    u32 type; /* Type of event */
    u32 size; /* Size of attribute structure */
    u64 config; /* Type-specific configuration */

    union {
        u64 sample_period; /* Period of sampling */
        u64 sample_freq; /* Frequency of sampling */
    };

    u64 sample_type; /* Specifies values included in sample */
    u64 read_format; /* Specifies values returned in read */

    u64 disabled : 1, /* off by default */
        inherit : 1, /* children inherit it */
        pinned : 1, /* must always be on PMU */
        exclusive : 1, /* only group on PMU */
        exclude_user : 1, /* don't count user */
        exclude_kernel : 1, /* don't count kernel */
        exclude_hv : 1, /* don't count hypervisor */
        exclude_idle : 1, /* don't count when idle */
        mmap : 1, /* include mmap data */
        comm : 1, /* include comm data */
        freq : 1, /* use freq, not period */
        inherit_stat : 1, /* per task counts */
        enable_on_exec : 1, /* next exec enables */
        task : 1, /* trace fork/exec */
        watermark : 1, /* wakeup_watermark */
        precise_ip : 2, /* skid constraint */
        mmap_data : 1, /* non-exec mmap data */
        sample_id_all : 1, /* sample_type all events */
        exclude_host : 1, /* don't count in host */
        exclude_guest : 1, /* don't count in guest */
        mmap2 : 1, /* include mmap with inode data */
        comm_exec : 1, /* flag comm events that are due to exec */
};
    
```

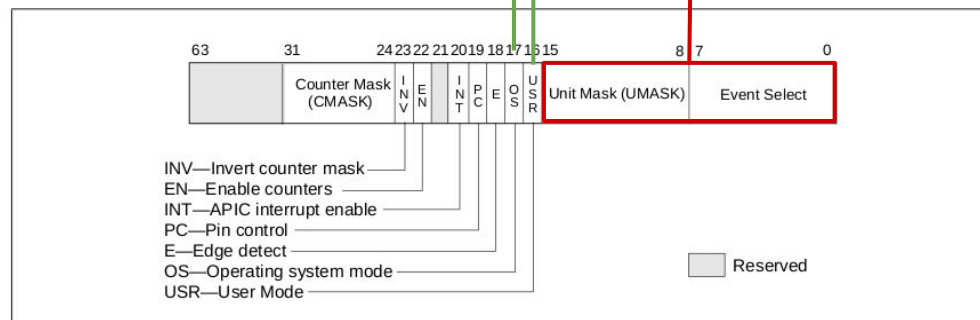


Figure 18-1. Layout of IA32_PERFEVTSELx MSRs

Implementation: Predict Raspberry Pi CPU Temp

- Last term
- More simple to understand before moving on to more complex task
- Subset of the main problem
 - PMU initialization
 - Forking and inherited monitoring
 - Python data collection manager

Hardware: Raspberry Pi 3b

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
 - Implements the **Arm Cortex A53** processor
 - Arm Cortex A53 implements the PMU
- ARM Cortex A53 PMU provides six counters
 - + dedicated cycle counter
- Each counter can count any of the events available in the processor.



Goal: Correlate microarchitectural events to temperature differential

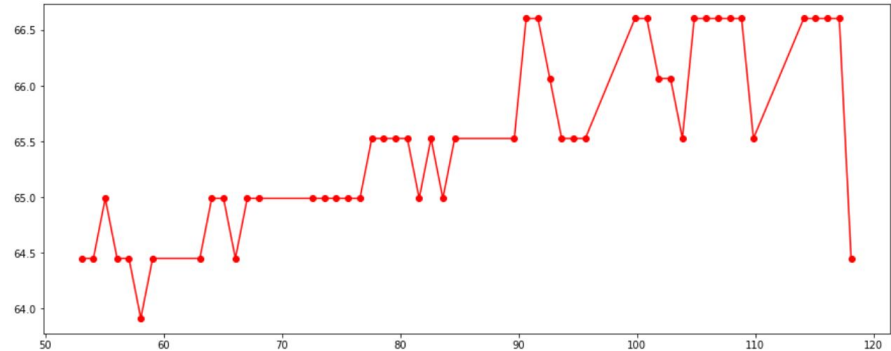
Methodology - Data collection

- Generate random workloads
- For each workload, monitor events
- Simultaneously monitor core temperature

Methodology - Regression

- Normalize event counts
- Ridge regression across normalized data

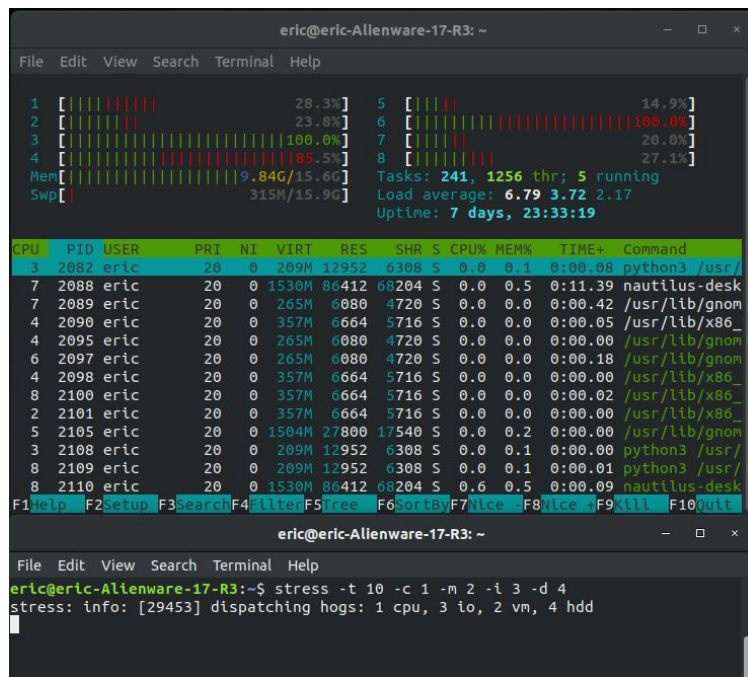
Temperature (C) over time (s)



Workload generation: Linux stress

Linux stress is a tool that launches subprocesses of specific types for a specified period of time, generating a work load that can be used to test a system.

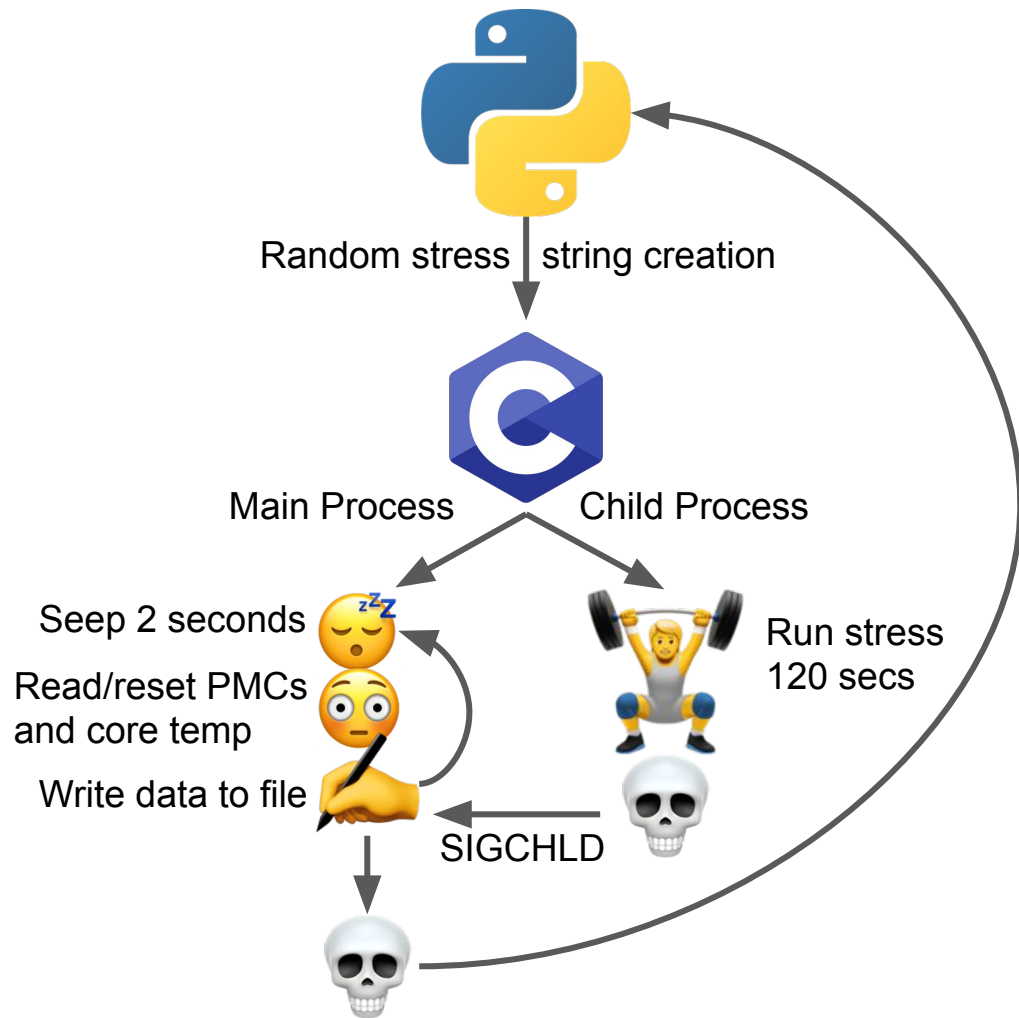
Opt	Meaning
-t N	Run for N seconds
-c N	Spawn N workers spinning on sqrt ()
-m N	spawn N workers spinning on malloc()/free()
-i N	Spawn N workers spinning on sync()
-d N	Spawn N workers spinning on write()/unlink()



```
eric@eric-Alienware-17-R3: ~  
File Edit View Search Terminal Help  
1 [|||||] 28.3% 5 [|||||] 14.9%  
2 [|||||] 23.8% 6 [|||||] 100.0%  
3 [|||||] 100.0% 7 [|||||] 20.0%  
4 [|||||] 100.5% 8 [|||||] 27.1%  
Mem [|||||] 9.84G/15.6G Tasks: 241, 1256 thr; 5 running  
Swp [|||||] 315M/15.9G Load average: 6.79 3.72 2.17  
Uptime: 7 days, 23:33:19  
  
CPU PID USER PRI NI VIRT RES SHR S CPU% MEM% TIME+ Command  
3 2082 eric 20 0 209M 12952 6308 S 0.0 0.1 0:00.08 python3 /usr/  
7 2088 eric 20 0 1530M 86412 68204 S 0.0 0.5 0:11.39 nautilus-desk  
7 2089 eric 20 0 265M 6080 4720 S 0.0 0.0 0:00.42 /usr/lib/gnom  
4 2090 eric 20 0 357M 6664 5716 S 0.0 0.0 0:00.05 /usr/lib/x86_  
4 2095 eric 20 0 265M 6080 4720 S 0.0 0.0 0:00.00 /usr/lib/gnom_  
6 2097 eric 20 0 265M 6080 4720 S 0.0 0.0 0:00.18 /usr/lib/gnom_  
4 2098 eric 20 0 357M 6664 5716 S 0.0 0.0 0:00.00 /usr/lib/x86_  
8 2100 eric 20 0 357M 6664 5716 S 0.0 0.0 0:00.02 /usr/lib/x86_  
2 2101 eric 20 0 357M 6664 5716 S 0.0 0.0 0:00.00 /usr/lib/x86_  
5 2105 eric 20 0 1504M 27800 17540 S 0.0 0.2 0:00.00 /usr/lib/gnom_  
3 2108 eric 20 0 209M 12952 6308 S 0.0 0.1 0:00.00 python3 /usr/  
8 2109 eric 20 0 209M 12952 6308 S 0.0 0.1 0:00.01 python3 /usr/  
8 2110 eric 20 0 1530M 86412 68204 S 0.0 0.5 0:00.09 nautilus-desk  
F1 Help F2 Setup F3 Search F4 Filter F5 Tree F6 Sort By F7 Nice F8 Nice + F9 Kill F10 Quit  
eric@eric-Alienware-17-R3: ~  
File Edit View Search Terminal Help  
eric@eric-Alienware-17-R3:~$ stress -t 10 -c 1 -m 2 -i 3 -d 4  
stress: info: [29453] dispatching hogs: 1 cpu, 3 io, 2 vm, 4 hdd
```


Data collection workflow

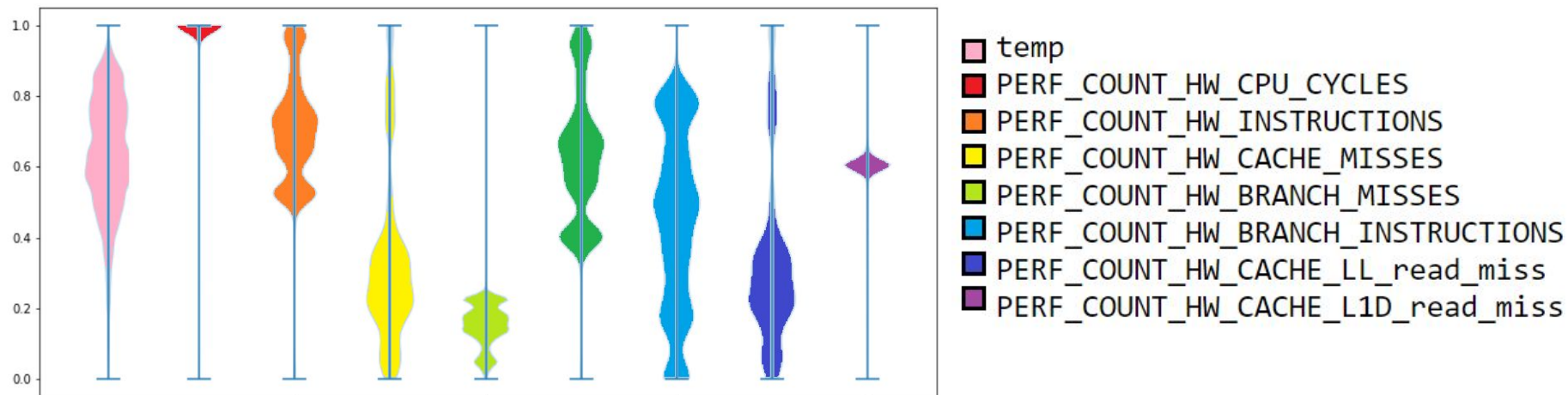
1. Python generates stress executable strings with random set of workers
2. C program takes stress string as input
3. C sets up a SIGCHLD listener that will terminate the program
4. C program forks a child process
 - a. The child process runs the stress workload
 - b. The parent process sleeps for 1 second and then records and resets the PMCs
5. Exit of child processes results in the termination of program.
6. Python feeds the next stress call in



Events Monitored

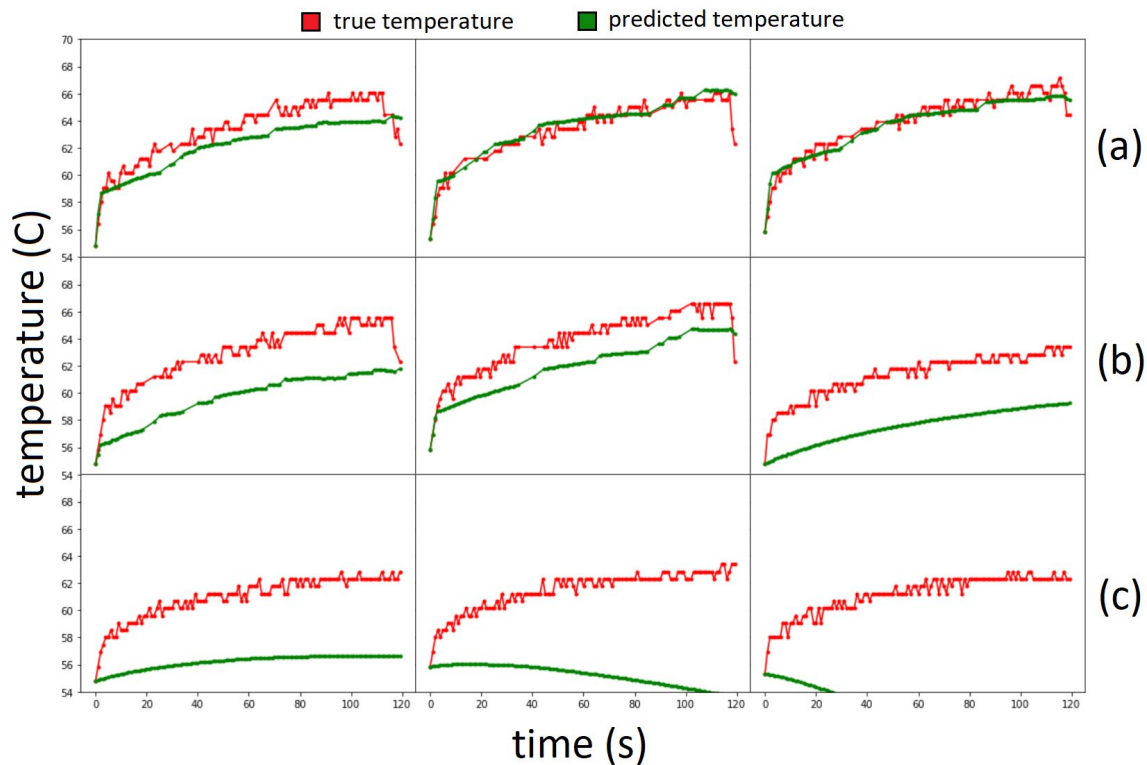
Event Name	Description
PERF_COUNT_HW_CPU_CYCLES	Total cycles. Be wary of what happens during CPU frequency scaling.
PERF_COUNT_HW_INSTRUCTIONS	Retired instructions. Be careful, these can be affected by various issues, most notably hardware interrupt counts
PERF_COUNT_HW_CACHE_MISSES	Cache misses. Usually 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

Normalized event count distributions



Results: Ridge regression, $\alpha=1$

	coef
temp	-0.281332
PERF_COUNT_HW_CPU_CYCLES	-0.062500
PERF_COUNT_HW_INSTRUCTIONS	0.161314
PERF_COUNT_HW_CACHE_MISSES	0.030964
PERF_COUNT_HW_BRANCH_MISSES	0.229393
PERF_COUNT_HW_BRANCH_INSTRUCTIONS	0.230638
PERF_COUNT_HW_CACHE_LL_read_miss	0.272127
PERF_COUNT_HW_CACHE_L1D_read_miss	0.030981



So there you have it

Using PMU events to predict a temperature

But wait!

- **How were the event chosen?** They seemed like a good idea...
- **Can we be sure the selected events were the best for this task?** No, not at all
- **How do we find the best events to monitor for a specific task?** Good question
- **Why can't we monitor all events and perform dimensionality reduction?**

PMU Based Program Classification on the 6th Gen Intel Core i7 (Skylake)

The multiplexing problem

Intel Skylake Architecture Events

- Architecture code name: Skylake
- 4 physical cores / 8 logical cores
 - Hyperthreading
 - Can turn hyperthreading
- Complex PMU
 - 8 Skylake event PMCs per core if hyperthreading is disabled in the BIOS
 - 4 PMCs per core if hyperthreading is enabled (standard)
 - [Lots of events!](#)

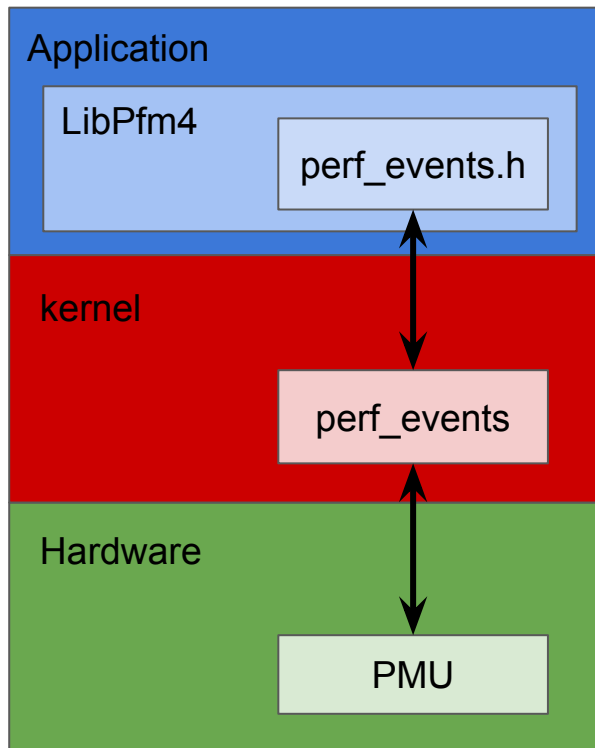


LibPfm4: Configuring events

Wrapper around `perf_events` that maintains list of available events and configurations for many architectures.

Given an “event string” input it will return a structure what contains the `perf_event_attr` structure.

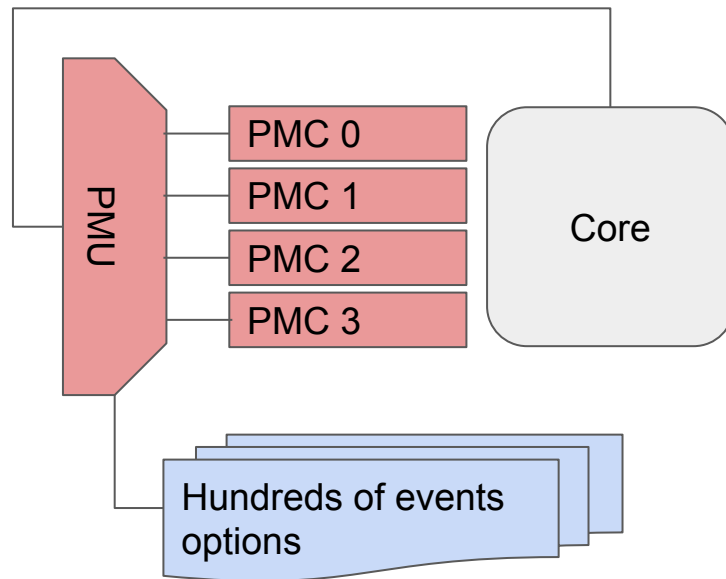
This structure can be fed directly into the `perf_event_open` system call.



The fundamental problem

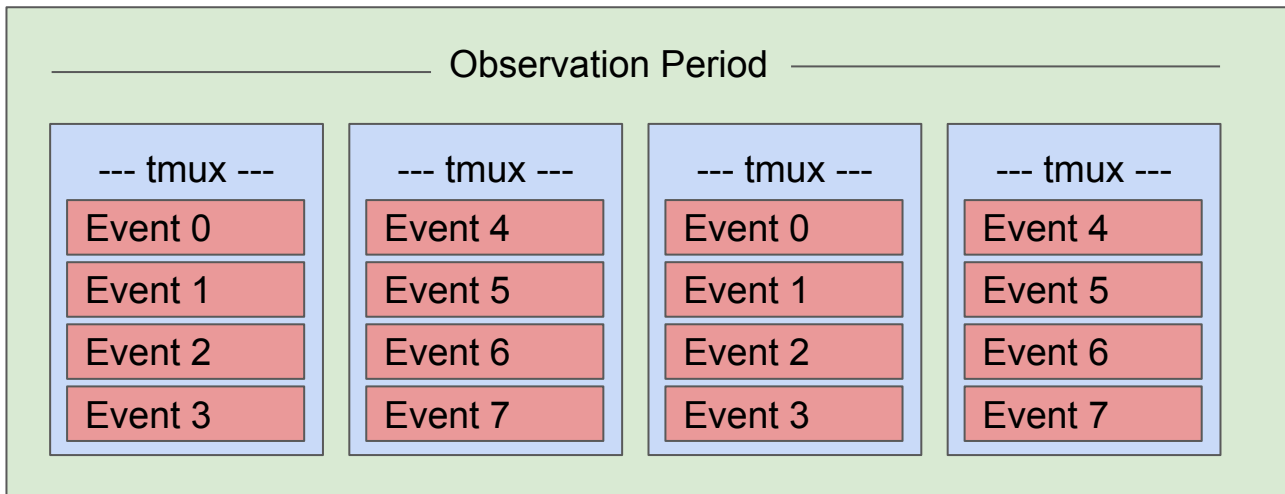
There are hundreds of events to monitor but only 4 counters per core.

This means that only up to 4 events can be monitored at any given time.



Temporal event multiplexing

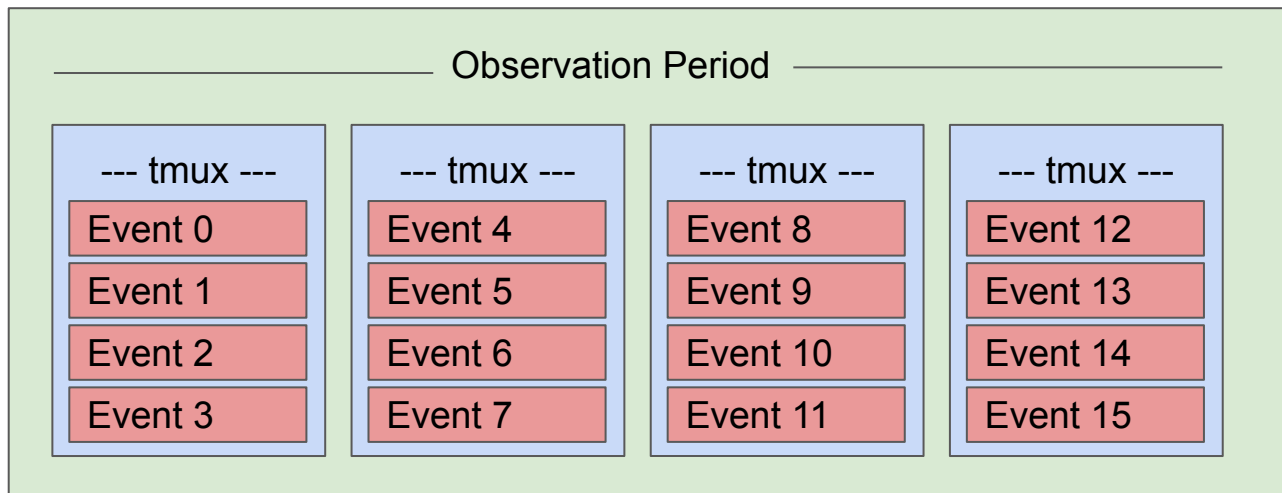
Swapping events on and off of the PMU in order to gather data on more events than there are performance counters. It is fundamentally a false view of reality since you miss all events that occur when that event is not programmed on the PMU. It is possible to normalize by capturing the amount of time an event was on the PMU during the observation period.



Problems with temporal multiplexing

Low Resolution

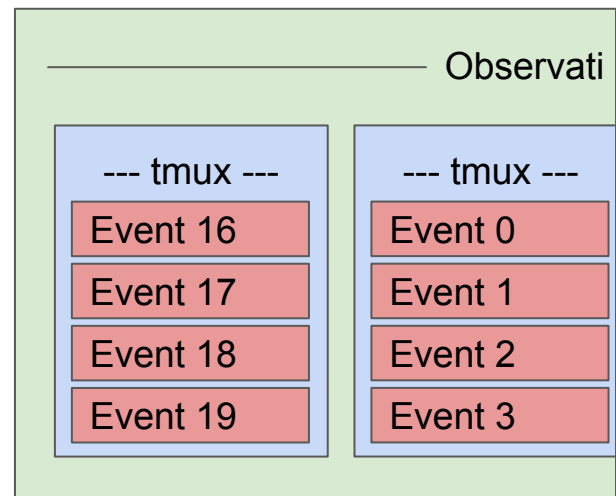
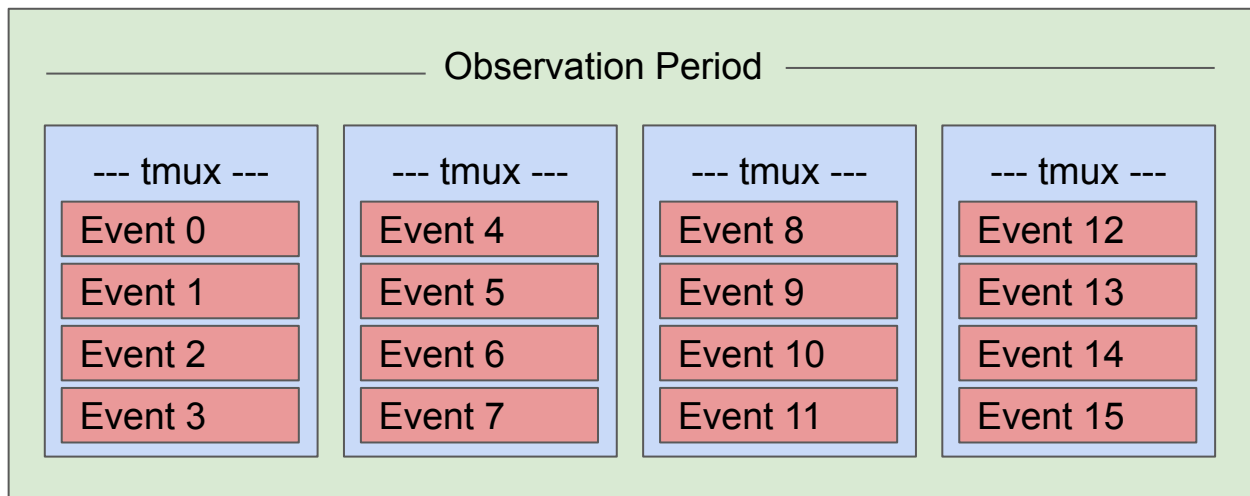
The fewer number of times events are scheduled on the PMU during an observation period the more exposed they are to temporal anomalies and the less accurately they reflect the reality of what is happening on the processor.



Problems with temporal multiplexing

Missing data in observations

If there are more events to be monitored than there are multiplexing periods in the observation period then it is possible for events to never be scheduled on the PMU during an period.



The Multiplexing Problem

- Multiplexing is useful in that it allows to gather, some level of statistical certainty, information on event occurrence about more events than there are available PMCs
- However, the more events there are the less precision we get.
- The observation window can be extended to allow for more event cycles, but then we begin aliasing on specific program functionality.
- If $\#events > PMCs/tmux$ then observations will be missing events entirely

Program Classification

- **Number of Events:** 230
- **PMC on Core:** 4
- **Tmux:** 4 ms
- **Observation Period:** 100 ms

How do we set up an experiment that determines the best combination of events to classify programs while avoiding the pitfalls of the multiplexing problem

What are the programs we are going to distinguish between?

- `stress -t 100 -d 1`
- `stress -t 100 -i 1`
- `stress -t 100 -i 1 -d 1`
- `stress -t 100 -m 1`
- `stress -t 100 -m 1 -d 1`
- `stress -t 100 -m 1 -i 1`
- `stress -t 100 -m 1 -i 1 -d 1`
- `stress -t 100 -c 1`
- `stress -t 100 -c 1 -d 1`
- `stress -t 100 -c 1 -i 1`
- `stress -t 100 -c 1 -i 1 -d 1`
- `stress -t 100 -c 1 -m 1`
- `stress -t 100 -c 1 -m 1 -d 1`
- `stress -t 100 -c 1 -m 1 -i 1`
- `stress -t 100 -c 1 -m 1 -i 1 -d 1`
- Each of the stress options either on or off
- Creates 15 programs to distinguish between
- Interesting because we know problems have a lot in common since they are different combinations of each other
- Each program runs for 100 seconds
- Since the observation period will be 100 ms, there will be 1,000 observations per program
- Total of ~15,000 observations

Learning Method: Event Forest

Data Collection Methodology

Sample the available event list without replacement N times to create N decision trees.

Data Collection Parameters

- Number of trees = 100
- Observation Period = 100 ms
- Min events = 4
- Max events = 10
 - $.1/((10/4)*.004) = 10$ rounds on PMU/event

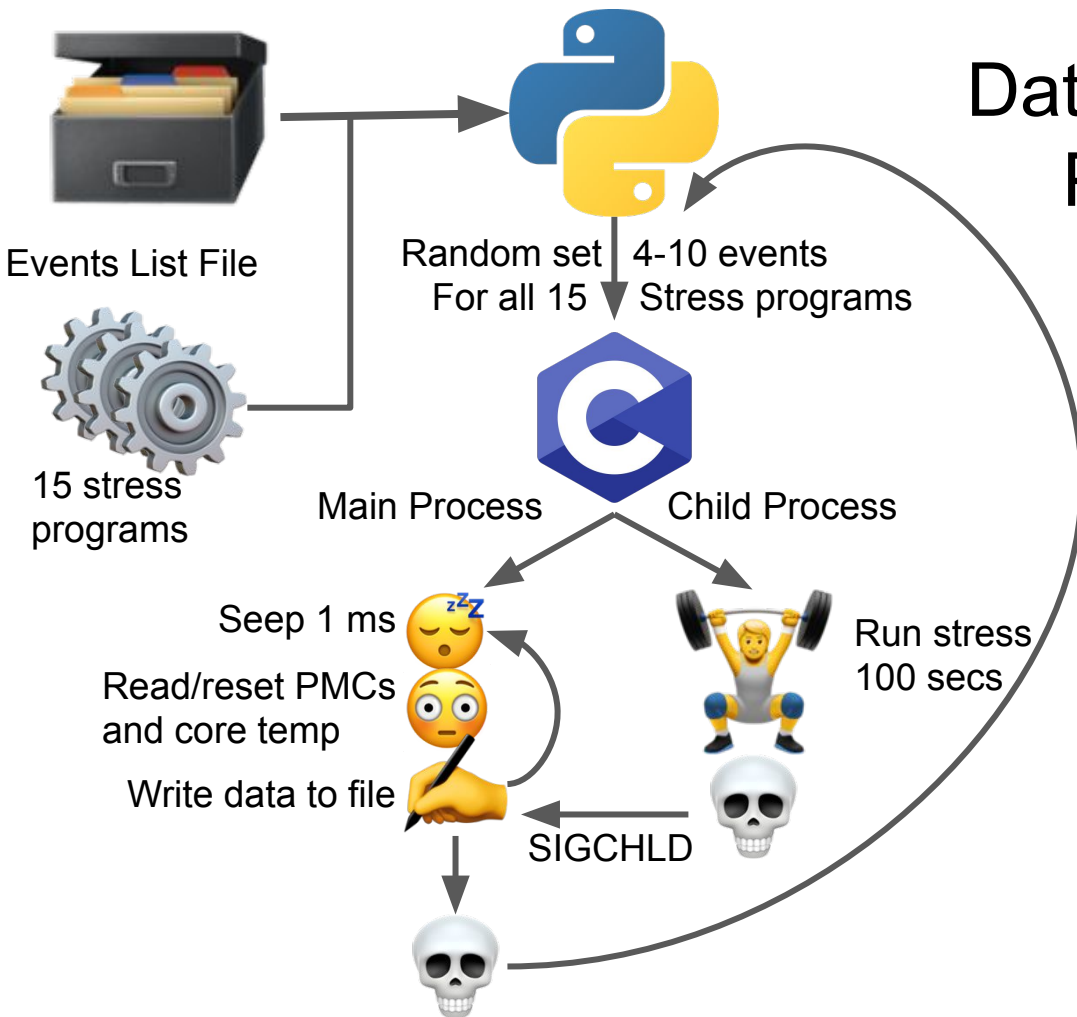
Learning Methodology

Apply standard decision tree algorithm to each tree and find get the best performing tree.

Training Hyperparameters

- Max depth = 8
- Min points = 64
- Impurity Metric = Gini

Data Collection: Program classification



1. Python gathers all events from valid event list and creates 100 sets of 4 to 10 events.
2. Python gathers the names of the different programs and manages the running of the programs against the specific event sets.
3. C program, using `perf_events` and `LibPfm4` generates the PMU configurations and loads the PMU.
4. C program monitors and records events just as last time, only associating events with a program instead of a temperature change.

Classifier Training: Decision Tree Set

1. Pandas is used to load and combine data files.
2. ScikitLearn applies decision tree algorithm to data in training dataset generating trees.
3. ScikitLearn runs test sets against built trees and outputs performance results.
4. The best tree is selected.



Raw PMU data

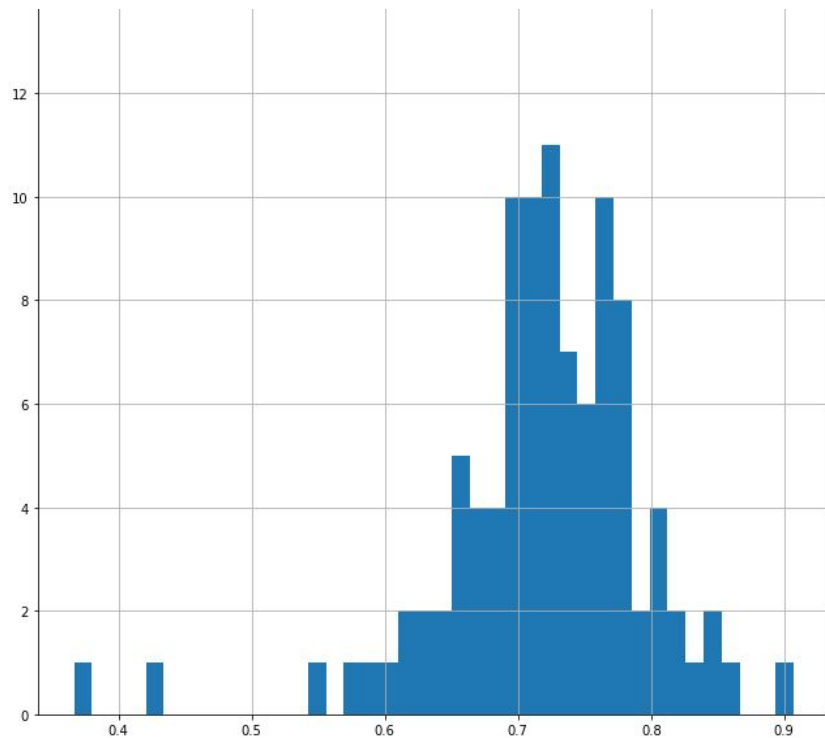


Individual tree data

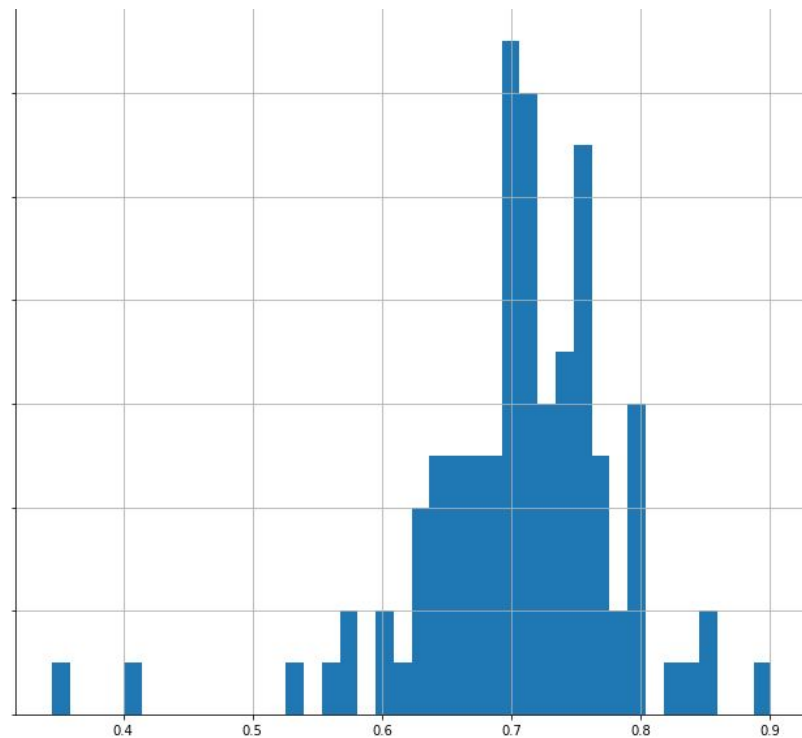


Results: Tree 1 - 100 accuracy

Training Accuracy



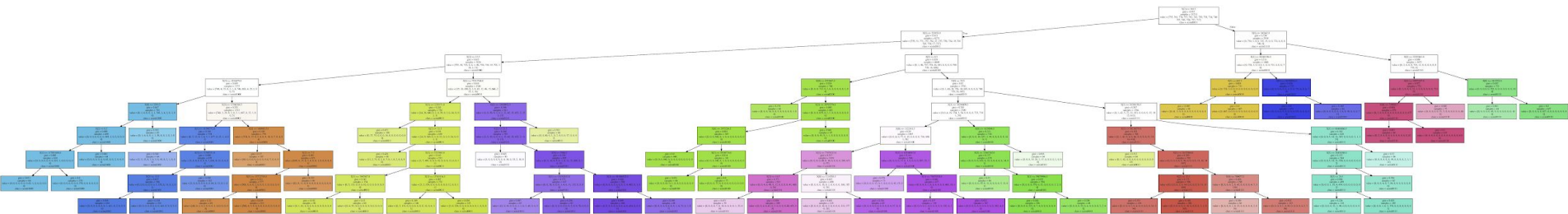
Testing Accuracy



Best Tree: 66

Tree Index	Event Name	Description
0	DTLB_STORE_MISSES:WALK_ACTIVE	Counts cycles when at least one PMH (Page Miss Handler) is busy with a page walk for a store.
1	INST_RETIRED:PREC_DIST	A version of INST_RETIRED that allows for a more unbiased distribution of samples across instructions retired. It utilizes the Precise Distribution of Instructions Retired (PDIR) feature to mitigate some bias in how retired instructions get sampled.
2	MEM_LOAD_MISC_RETIRED:UC	Retired instructions with at least 1 uncacheable load or lock.
3	UOPS_EXECUTED:THREAD	Number of uops to be executed per-thread each cycle.
4	TX_EXEC:MISC2	Unfriendly TSX abort triggered by a vzeroupper instruction.
5	TX_MEM:ABORT_HLE_STORE_TO_ELIDED_LOCK	Number of times a TSX Abort was triggered due to a non-release/commit store to lock.

Best Tree: 66



Recap

- Introduced the PMU
- Introduced tool to make working with the PMU easier
 - Perf_events for interacting with the PMU with C
 - LibPfm4 for generating configurations to be fed to perf_events
- Demonstrated PMU data can be used in data science application
 - Regression on pre selected events
 - Classification on all available events
- Identified a unique problem, uncommon in standard data science workflows
 - The multiplexing problem doesn't allow for all features in a dataset to be gathered simultaneously
- Proposed a simplistic form of dimensionality reduction to address the multiplexing problem

Gratitude to the open source community

I do not possess the skill or the time to have been able to move this project forward without the wonderful tools that I was able to use.

It is a wonderful time to be a programmer and that is thanks to the open source community in a major way.

- Ubuntu Linux
- DebianStretch Linux
- Perf_events
- LibPfm4
- Python3
- ScikitLearn
- Pandas
- Matplotlib
- Stress

Thank you