Announcements

- Will post Project schedule
- HW#11 will be posted
CANbus

• Automotive. Introduced by BOSCH, 1983
• One of OBD-II protocols
• differential, 2 wires, 1MBps important things like engine control
• single wire, slower cheaper, hvac, radio, airbags
CANbus Protocol

• id, length code, up to 8 bytes of data id (usually 11 or 29 bits) type and who is sending it. Also priority (lower is higher) length is 4 bits. some always send 8 and pad with zeros

• Type is inferred from id. Can be things like engine RPM, etc

• DBC database has the ids and values. ASCII text database, hard to get legally.
• Dominant/Recessive. Message with lowest ID wins arbitration.

• CAN-FD – extended version with larger sizes
Can4linux – open("/dev/can0"); read(); write();
External project?

SocketCAN – contributed by Volkswagen. In kernel. Uses socket interface. /Documentation/networking/can.txt
CANbus on Pi

- No, but you can get SPI or I2C adapters
Measuring Power and Energy

- Sense resistor or Hall Effect sensor gives you the current
- Sense resistor is small resistor. Measure voltage drop. Current $V=IR$ Ohm’s Law, so $V/R=I$
- Voltage drops are often small (why?) so you made need to amplify with instrumentation amplifier
- Then you need to measure with A/D converter
- $P = IV$ and you know the voltage
- How to get Energy from Power?
Definitions

People often say Power when they mean Energy

- Dynamic Power – only consumed while computing
- Static Power – consumed all the time.
  Sets the lower limit of optimization
Units

- **Energy** – Joules, kWh (3.6MJ), Therm (105.5MJ), 1 Ton TNT (4.2GJ), eV ($1.6 \times 10^{-19}$ J), BTU (1055 J), horsepower-hour (2.68 MJ), calorie (4.184 J)
- **Power** – Energy/Time – Watts (1 J/s), Horsepower (746W), Ton of Refrigeration (12,000 Btu/h)
- **Volt-Amps** (for A/C) – same units as Watts, but not same thing
- **Charge** – mAh (batteries) – need voltage to convert to Energy
CPU Power and Energy
CMOS Transistors

N–MOSFET

Source
Gate
Drain

p–substrate

P–MOSFET

Source
Gate
Drain

n–well
CMOS Dynamic Power

- \( P = C \Delta V V_{dd} \alpha f \)

  Charging and discharging capacitors big factor
  \((C \Delta V V_{dd})\) from \(V_{dd}\) to ground

  \(\alpha\) is activity factor, transitions per clock cycle

  \(f\) is frequency

- \(\alpha\) often approximated as \(\frac{1}{2}\), \(\Delta V V_{dd}\) as \(V_{dd}^2\) leading to

  \(P \approx \frac{1}{2} CV_{dd}^2 f\)

- Some pass-through loss (\(V\) momentarily shorted)
CMOS Dynamic Power Reduction

How can you reduce Dynamic Power?

- Reduce $C$ – scaling
- Reduce $V_{dd}$ – eventually hit transistor limit
- Reduce $\alpha$ (design level)
- Reduce $f$ – makes processor slower
CMOS Static Power

- Leakage Current – bigger issue as scaling smaller. Forecast at one point to be 20-50% of all chip power before mitigations were taken.

- Various kinds of leakage (Substrate, Gate, etc)

- Linear with Voltage: $P_{static} = I_{leakage} V_{dd}$
Leakage Mitigation

- SOI – Silicon on Insulator (AMD, IBM but not Intel)
- High-k dielectric – instead of SO2 use some other material for gate oxide (Hafnium)
- Transistor sizing – make only critical transistors fast; non-critical can be made slower and less leakage prone
- Body-biasing
- Sleep transistors
Total Energy

- \( E_{\text{tot}} = [P_{\text{dynamic}} + P_{\text{static}}]t \)
- \( E_{\text{tot}} = [(C_{\text{tot}}V_{dd}^2\alpha f) + (N_{\text{tot}}I_{\text{leakage}}V_{dd})]t \)
Delay

- \[ T_d = \frac{C_L V_{dd}}{\mu C_{ox}(\frac{W}{L})(V_{dd}-V_t)} \]

- Simplifies to \[ f_{MAX} \sim \frac{(V_{dd}-V_t)^2}{V_{dd}} \]

- If you lower f, you can lower \( V_{dd} \)
Thermal Issues

- Temperature and Heat Dissipation are closely related to Power
- If thermal issues, need heatsinks, fans, cooling
Metrics to Optimize

- Power
- Energy
- MIPS/W, FLOPS/W (don't handle quadratic V well)
- $Energy \times Delay$
- $Energy \times Delay^2$
Power Optimization

- Does not take into account time. Lowering power does no good if it increases runtime.
Energy Optimization

- Lowering energy can affect time too, as parts can run slower at lower voltages
Energy Delay – Watt/t*t

- Horowitz, Indermaur, Gonzalez (Low Power Electronics, 1994)

- Need to account for delay, so that lowering Energy does not make delay (time) worse

- Voltage Scaling – in general scaling low makes transistors slower

- Transistor Sizing – reduces Capacitance, also makes transistors slower
• Technology Scaling – reduces V and power.

• Transition Reduction – better logic design, have fewer transitions
  Get rid of clocks? Asynchronous? Clock-gating?

• Example with inverse ED (higher better):

<table>
<thead>
<tr>
<th></th>
<th>SPEC</th>
<th>Power</th>
<th>SPEC*SPEC/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21064</td>
<td>155</td>
<td>30W</td>
<td>800</td>
</tr>
<tr>
<td>PPC603</td>
<td>80</td>
<td>3W</td>
<td>2100</td>
</tr>
</tbody>
</table>
Energy Delay Squared—$E*t^2$

- Martin, Nyström, Pénzes – Power Aware Computing, 2002

- Independent of Voltage in CMOS

- $E_t$ can be misleading
  $E_a=2E_b$, $t_a=t_B/2$
  Reduce voltage by half, $E_a=E_a/4$, $t_a=2t_a$, $E_a=E_b/2$, $t_a=t_b$

- Can have arbitrary large number of delay terms in Energy
product, squared seems to be good enough
## Power and Energy Concerns

### Table 1: ATLAS 300x300 DGEMM (Matrix Multiply)

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor</th>
<th>Cores</th>
<th>Frequency</th>
<th>Idle</th>
<th>Load</th>
<th>Time</th>
<th>Total Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raspberry Pi</td>
<td>ARM 1176</td>
<td>1</td>
<td>700MHz</td>
<td>3.0W</td>
<td>3.3W</td>
<td>23.5s</td>
<td>77.6J</td>
</tr>
<tr>
<td>Gumstix Overo</td>
<td>Cortex-A8</td>
<td>1</td>
<td>600Mhz</td>
<td>2.6W</td>
<td>2.9W</td>
<td>27.0s</td>
<td>78.3J</td>
</tr>
<tr>
<td>Beagleboard</td>
<td>Cortex-A8</td>
<td>1</td>
<td>800MHz</td>
<td>3.6W</td>
<td>4.5W</td>
<td>19.9s</td>
<td>89.5J</td>
</tr>
<tr>
<td>Pandaboard</td>
<td>Cortex-A9</td>
<td>2</td>
<td>900MHz</td>
<td>3.2W</td>
<td>4.2W</td>
<td>1.52s</td>
<td>6.38J</td>
</tr>
<tr>
<td>Chromebook</td>
<td>Cortex-A15</td>
<td>2</td>
<td>1.7GHz</td>
<td>5.4W</td>
<td>8.1W</td>
<td>1.39s</td>
<td>11.3J</td>
</tr>
</tbody>
</table>
Questions

• Which machine consumes the least amount of energy? (Pandaboard)

• Which machine computes the result fastest? (Chromebook)

• Chromebook is a laptop so also includes display and wi-fi

• Consider a use case with an embedded board taking a picture once every 20 seconds and then performing a
300x300 matrix multiply transform on it. Could all of the boards listed meet this deadline?
No, the Raspberry Pi and Gumstix Overo both take longer than 20s and the Beagleboard is dangerously close.

- Assume a workload where a device takes a picture once a minute then does a 300x300 matrix multiply (as seen in Table 1). The device is idle when not multiplying, but under full load when it is. Over an hour, what is the energy usage of the Chromebook? What is the energy usage of the Gumstix?
Chromebook per minute: \((1.39s \times 8.1W) + (58.61s \times 5.4W) = 327.75J\)

Chromebook per hour: \(327.75J \times 60 = 19.7kJ\)

Gumstix per minute: \((27s \times 2.9W) + (33s \times 2.6W) = 164.1J\)

Gumstix per hour: \(164.1J \times 60 = 9.8kJ\)
Easy ways to reduce Power Usage
DVFS

- Voltage planes – on CMP might share voltage planes so have to scale multiple processors at a time
- DC to DC converter, programmable.
- Phase-Locked Loops. Orders of ms to change. Multiplier of some crystal frequency.
- Senger et al ISCAS 2006 lists some alternatives. Two phase locked loops? High frequency loop and have programmable divider?
• Often takes time, on order of milliseconds, to switch frequency. Switching voltage can be done with less hassle.
When can we scale CPU down?

- System idle
- System memory or I/O bound
- Poor multi-threaded code (spinning in spin locks)
- Thermal emergency
- User preference (want fans to run less)
Introduction to Performance Analysis
What is Performance?

- Getting results as quickly as possible?
- Getting *correct* results as quickly as possible?
- What about Budget?
- What about Development Time?
- What about Hardware Usage?
- What about Power Consumption?
Know Your Limitation

- CPU Constrained
- Memory Constrained (Memory Wall)
- I/O Constrained
- Thermal Constrained
- Energy Constrained
Performance Optimization Cycle

- Develop Code
- Functionally Complete/Correct Code
- Measure
- Analyze
- Modify / Tune
- Functionally Complete/Correct/Optimized Code
- Usage / Production
Wisdom from Knuth

“We should forget about small efficiencies, say about 97% of the time:

premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified” — Donald Knuth
Amdahl’s Law

<table>
<thead>
<tr>
<th>Speed up Blue 100x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed up Red 2x</td>
</tr>
</tbody>
</table>

Time
Measuring Time

• Already talked about Power, but other aspect is speed (time)

• time command

• Reports real (wall-clock), user (used by program), sys (kernel)

• In virtualized systems wall-clock time might become meaningless
• Timers, rdtsc?

• When can user time exceed real? (multi-threaded)

• When can user+sys be less than real? (If something else is using the system)

• Waiting on I/O and Interrupts count as sys time.
Using “time”

```
vince@rasp-pi5 ~/research/libpfm4/examples $ time /bin/ls check_events check_events.o showevtinfo showevtinfo.o check_events.c Makefile showevtinfo.c

real  0m0.018s
user  0m0.010s
sys   0m0.000s
```

What do they mean? Can real be higher than user? Can user be more than real? Is it deterministic (will it vary run
to run)
What are Hardware Performance Counters?

- Registers on CPU that measure low-level system performance
- Available on most modern CPUs; increasingly found on GPUs, network devices, etc.
- Low overhead to read
Low-level interface

• on x86: MSRs

• ARM: CP15 system control register
CP15 registers in Pi

- BCM2835 (Original Pi)
  - 3 counters available (1 cycle counter, 2 generic)
  - 25 events
  - No way to specify kernel vs user
  - On Raspberry Pi original overflow interrupt not connected

- BCM2836 (Pi2)
  - The ARM-Cortex A7 has 5 counters
  - Can specify kernel, user
- Overflow works

- BCM2837 (Pi3)
  - The ARM-Cortex A53 has 7 counters
  - Can specify kernel, user
  - Overflow works
CP15 Interface

- use mcr, mrc to move values in/out
  MRC p15,0,Rt,c9,c12,0
  MCR p15,0,Rt,c9,c12,0

- Two EVNTCNT registers

- Cycle Counter register

- Two Event Config registers

- Count enable set/clear, count interrupt enable/clear,
overflow, software increment

- PMU management registers
- in general only privileged access (why) but can be configured to let users access.
Hardware Performance Counters: The Operating System Interface
Operating System Interface

A typical operating system performance counter interface will provide the following:

• A way to select which events are being monitored

• A way to start and stop counting

• A method of reading counter results when finished, and

• If the CPU supports notification on counter overflow, some mechanism for passing on overflow information
Operating System Interface

Some operating systems provide additional features:

- Event scheduling: often there are limitations on which events can go into which counters,

- Multiplexing: the OS can hide the fact that only a limited number of counters are available by swapping events in and out and extrapolating counts using time accounting,

- Per-thread counting: by loading and saving counter
values at context switch time a count specific to a process can be achieved,

- Attaching to a process: counts can be taken from an already running process, and

- Per-cpu counting: as with per-thread counting, counts can be accumulated per-cpu.
Older Linux Interfaces

- Historical – typically just exported msrs
- Oprofile – only does profiling
- Perfctr – good but required kernel patch
- Perfmon2 – was making headway until perf_event came from nowhere and became official
perf_event

- Developed from scratch in 2.6.31 by Molnar and Gleixner
- Everything in the kernel
- `perf_event_open()` syscall (manpage still under development)
- `perf_event_attr` structure with 40 complex interdependent parameters
- `ioctl()` system call to enable/disable
• read() system call to read values
• can gather sampled data in circular buffer
• can get signal on overflow or full buffer
perf_event Generalized Events

• perf_event provides support for “common” generalized events

• makes things easier for user at expense of papering over the differences between events

• events need to be validated to make sure they are providing useful results
perf_event Generalized Events Issues

- Which event to choose (Nehalem)
- From 2.6.31 to 2.6.35 AMD “branches” was taken not total
- Nehalem L1 DCACHE reads.
  PAPI uses L1D_CACHE_LD:MESI;
  perf uses MEM_INST_RETIRED:LOADS
perf_event Event Scheduling

- Some events have hardware constraints. Can only be in one counter
- You can do this scheduling in userspace; lets the algorithm be changed more easily
- Scheduling can be expensive; do so at event start can slow things down.
perf_event Multiplexing

- You may wish to measure more events simultaneously than hardware can support (NMI watchdog may steal one too)

- perf_event supports this in-kernel (you can also do this in userspace)

- there are various ways to try to ensure good statistical results. in kernel you have to trust the kernel programmers.
perf_event Event Names

- Event names are provided in the hardware manuals, but can be inconsistent
- Traditionally used libraries to provide names. libpfm4
- perf tool is starting to provide own list of events (they refuse to link libpfm4) that are based on a hybrid of libpfm4 and kernel names
- Also some event names are provided by the kernel under /sys
perf_event Software Events

- perf_event provides internal kernel events through same interface
- page-fault, task-clock, cpu-clock, etc.
**perf_event Perf Tool**

- Included with kernel source code
- Tied to kernel, but backwards compatible
- Most kernel devs use this rather than outside tools
- `apt-get install linux-perf (new)` or `linux-tools (old)`
perf

Based on a tutorial found here: https://perf.wiki.kernel.org/index.php/Tutorial
Lists available events

List of pre-defined events (to be used in `-e`):

<table>
<thead>
<tr>
<th>Hardware events</th>
<th>Software events</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu-cycles OR cycles</td>
<td>cpu-clock</td>
</tr>
<tr>
<td>instructions</td>
<td>task-clock</td>
</tr>
<tr>
<td>cache-references</td>
<td>page-faults OR faults</td>
</tr>
<tr>
<td>cache-misses</td>
<td>minor-faults</td>
</tr>
<tr>
<td>branch-instructions OR branches</td>
<td>major-faults</td>
</tr>
<tr>
<td>branch-misses</td>
<td>context-switches OR cs</td>
</tr>
<tr>
<td>bus-cycles</td>
<td></td>
</tr>
</tbody>
</table>
vincent@arm:/class/ece571$ perf stat ./matrix_multiply
Matrix multiply sum: s=2766573402509.746094

Performance counter stats for './matrix_multiply':

1158.141036 task-clock # 0.999 CPUs utilized
  19 context-switches # 0.000 M/sec
  0 CPU-migrations # 0.000 M/sec
  1,633 page-faults # 0.000 M/sec
10,343,746,076 cycles # 0.893 GHz
  5,031,717 stalled-cycles-frontend # 0.05% frontend cycles idle
  9,521,135,479 stalled-cycles-backend # 92.05% backend cycles idle
1,176,286,814 instructions # 0.11 insns per cycle
  831,736 branch-misses # 8.09 stalled cycles per insn
137,835,961 branches # 11.898 M/sec
  831,736 branch-misses # 0.60% of all branches

11.591796875 seconds time elapsed
perf stat – Specifying Events

```bash
vince@arm:~/class/ece571$ perf stat -e instructions,cycles ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

1,174,788,622 instructions # 0.14 insns per cycle
8,346,588,065 cycles # 0.000 GHz

12.394775391 seconds time elapsed```

perf stat – Specifying Masks

:u is user, :k kernel
ARM Cortex A9 cannot specify this distinction (results shown here are x86)

```
vince@arm:~/class/ece571$ perf stat -e instructions,instructions:u ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

  950,526,051 instructions #  0.00  insns per cycle
  945,661,967 instructions:u  #  0.00  insns per cycle

1.052072277 seconds time elapsed```

libpfm4 – Finding All Event Names

./showevtinfo
Supported PMU models:
  [51, perf, "perf_events generic PMU"]
  [65, arm_ac8, "ARM Cortex A8"]
  [66, arm_ac9, "ARM Cortex A9"]
  [75, arm_ac15, "ARM Cortex A15"]
Detected PMU models:
  [51, perf, "perf_events generic PMU", 80 events, 1 max encoding, 0 counters, OS generic PMU]
  [66, arm_ac9, "ARM Cortex A9", 57 events, 1 max encoding, 2 counters, core PMU]
Total events: 254 available, 137 supported
...
#----------------------------
IDX     : 138412068
PMU name: arm_ac9 (ARM Cortex A9)
Name     : NEON_EXECUTED_INST
Equiv    : None
Flags    : None
Desc     : NEON instructions going through register renaming stage (approximate)
Code     : 0x74
#----------------------------
....
libpfm4 – Finding Raw Event Values

./check_events NEON_EXECUTED_INST
Supported PMU models:
[51, perf, "perf_events generic PMU"]
[65, arm_ac8, "ARM Cortex A8"]
[66, arm_ac9, "ARM Cortex A9"]
[75, arm_ac15, "ARM Cortex A15"]
Detected PMU models:
[51, perf, "perf_events generic PMU"]
[66, arm_ac9, "ARM Cortex A9"]
Total events: 254 available, 137 supported
Requested Event: NEON_EXECUTED_INST
Actual Event: arm_ac9::NEON_EXECUTED_INST
PMU : ARM Cortex A9
IDX : 138412068
Codes : 0x74
vinced@arm:~$ perf stat -e r74 ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

1 r74

11.303955078 seconds time elapsed
perf stat – multiplexing

```
perf stat -e instructions,instructions,branches,cycles,cycles ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

1,178,121,057 instructions # 0.12 insns per cycle [40.23%]
1,180,460,368 instructions # 0.12 insns per cycle [60.25%]
138,550,072 branches [80.09%]
9,999,614,616 cycles # 0.000 GHz [79.85%]
9,926,949,659 cycles # 0.000 GHz [20.17%]

11.214630127 seconds time elapsed
```

Note same event not same results, approximate because an estimate. Percentage shown is percentage event was active during run.
perf stat – all cores

vince@arm:~/class/ece571$ sudo perf stat -a ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

Performance counter stats for './matrix_multiply':

<table>
<thead>
<tr>
<th>Stat</th>
<th>Value</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>task-clock</td>
<td>24089.660644</td>
<td>100.00%</td>
</tr>
<tr>
<td>context-switches</td>
<td>105</td>
<td>0.000 M/sec</td>
</tr>
<tr>
<td>page-faults</td>
<td>1,641</td>
<td>0.000 M/sec</td>
</tr>
<tr>
<td>cycles</td>
<td>9,218,451,619</td>
<td>0.383 GHz</td>
</tr>
<tr>
<td>stalled-cycles-frontend</td>
<td>9,707,195</td>
<td>0.11%</td>
</tr>
<tr>
<td>stalled-cycles-backend</td>
<td>8,393,095,067</td>
<td>91.05%</td>
</tr>
<tr>
<td>instructions</td>
<td>1,193,164,945</td>
<td>0.13 insns per cycle</td>
</tr>
<tr>
<td>branches</td>
<td>139,913,572</td>
<td>5.808 M/sec</td>
</tr>
<tr>
<td>branch-misses</td>
<td>1,221,237</td>
<td>0.87% of all branches</td>
</tr>
</tbody>
</table>

12.040527344 seconds time elapsed

Run on all cores of system even if your process not running there. -a option. Need root permissions
perf record – sampling

```
vince@arm:~/class/ece571$ time ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094

real 0m10.747s
user 0m10.688s
sys  0m0.055s
vince@arm:~/class/ece571$ time perf record ./matrix_multiply
Matrix multiply sum: s=27665734022509.746094
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.454 MB perf.data (~19853 samples) ]

real 0m12.009s
user 0m11.797s
sys  0m0.203s
```

**perf record creates perf.data, use -o to specify output**
perf report – summary of recorded data

<table>
<thead>
<tr>
<th>Time (sec)</th>
<th>Function Call</th>
<th>Time (sec)</th>
<th>Function Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.62%</td>
<td>matrix_multiply matrix_multiply</td>
<td>0.38%</td>
<td>matrix_multiply [kernel.kallsyms].head.text</td>
</tr>
<tr>
<td>0.38%</td>
<td>matrix_multiply [kernel.kallsyms].head.text</td>
<td>0.00%</td>
<td>matrix_multiply ld-2.13.so</td>
</tr>
<tr>
<td>0.00%</td>
<td>matrix_multiply ld-2.13.so</td>
<td>0.00%</td>
<td>matrix_multiply [kernel.kallsyms]</td>
</tr>
</tbody>
</table>

Our benchmark is simple (only one function) so the profiled results are not that exciting.

The [k] indicates that profile happened while the kernel was running.
The annotated results show a branch and an add instruction accounting for 83% of profiles. Likely this is due to skid and the key instruction is the previous `vmla.f64` floating point multiply instruction. The processor just isn’t able to stop at the exact instruction when the interrupt comes in.