CPU Architecture
ASD Shared Memory HPC Workshop

June 2023

School of Computing
Australian National University
Canberra, Australia

Outline

Introduction

Performance Measurement and Modeling

Example Applications

Hardware Performance Counters

High Performance Microprocessors

Loop Optimization: Software Pipelining

Schedule - Day 1 - CPU Architecture

Time	Lecture Topics	Hands-On Exercise
9:00	Introduction and Course Overview	Logging on and timing things
10:30	*Morning Tea*	
11:00	Applications Hardware Performance Counters and Metrics	PAPI Hardware Performance Counters
12:30	*Lunch*	
13:30	Architectural Features	Pipelining
15:00	*Afternoon Tea*	
15:30	Loop optimization	Loop ordering and unrolling

Schedule - Day 2 - Vectorization and Cache

Time	Lecture Topics	Hands-On Exercise
9:00	SIMD operations	Vectorizing loops using SSE & NEON
10:30	*Morning Tea*	
11:00	Cache structure and organization	Instruction per cycle and cache misses
12:30	*Lunch*	
13:30	Multiprocessor cache organization	Matrix multiplication performance
15:00	*Afternoon Tea*	
15:30	Thread basics	False cache line sharing

Schedule - Day 3 - Multiprocessor Parallelism

Time	Lecture Topics	Hands-On Exercise
9:00	OS thread support Mutual exclusion	Non-determinism
10:30	*Morning Tea*	
11:00	Higher-level synchronization constructs Memory consistency	Implementing synchronization constructs / Deadlock
12:30	*Lunch*	
13:30	OpenMP	OpenMP
15:00	*Afternoon Tea*	
15:30	OpenMP Tasks	OpenMP Tasks

Schedule - Day 4 - Parallel Performance Optimization

Time	Lecture Topics	Hands-On Exercise
9:00	Non-uniform memory access	Using thread and memory placement on NUMA systems
10:30	*Morning Tea*	
11:00	Profiling	Profiling applications using VTune
12:30	*Lunch*	
13:30	C++11 Threads	Threads - C++
15:00	*Afternoon Tea*	
15:30	Lock-free synchronization Transactional memory	Lock and barrier performance

Schedule - Day 5 - Parallel Software Design

Time	Lecture Topics	Hands-On Exercise
9:00	Overview of parallel software patterns Finding concurrency	
10:30	*Morning Tea*	
11:00	Threading Building Blocks	Programming with TBB
12:30	*Lunch*	
13:30	Algorithmic structure patterns Program and data structure patterns	
15:00	*Afternoon Tea*	
15:30	Emerging paradigms and challenges	

Course Hardware - Specifications

NCI Gadi system - Intel Cascade Lake

- 2 x Intel Xeon Platinum 8274 (24-core) with HyperThreading, 3.2 GHz
- 32 KB 8-way L1 D-Cache, 1MB 16-way L2 D-Cache, 36 MB 11-way L3 Cache (shared), 64B line
- 196 GB DDR4 RAM

ARM system - Neoverse

- 32 Neoverse N1 cores, 2.6GHz (AWS Graviton2 instances: 16 vCPUs)
- 64 KB 4-way L1 D-Cache, 512 KB 8-way L2 Cache, 4 MB 16-way L3 Cache (shared)
- 32 GB RAM

More details @ https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake and https://en.wikichip.org/wiki/arm_holdings/microarchitectures/neoverse_n1

Course Hardware - Logging in

Follow the instructions provided at
https://github.com/ANU-HPC/sharedMemHPC_exercises/tree/main/systems.md

Outline

Introduction

Performance Measurement and Modeling

- Measuring Time
- Performance Modeling

Example Applications

Hardware Performance Counters

High Performance Microprocessors

Loop Optimization: Software Pipelining

Measuring Time

- Which time to use: wall time (elapsed time), or process time?
- Reliability issues (typical time slice interval is $t_S \approx 0.01s$):

time:	wall	process
timer resolution t_R:	high ✓	low ($= t_S$) ✗
timer call overhead t_C:	low ✓	high ✗
effect of time slicing / interrupts:	high ✗	lower ✓
appropriate timing interval t_I:	$< 1t_S$	$> 100t_S$

- Error in $t_I \leq | \pm 2t_R + t_C|$ (may be variability in t_C; $t_I \leq 2t_R + t_C$ safer)
  - how to minimize these effects?
- Estimating t_R from (differences between) repeated calls to a timer function:
  - 16e-06 0 5.0e-6 0 0 0 0 5.0e-6 0 0 0 0 5.0e-6 0 . . . : $t_R \approx 5e-6$ ($t_C \approx 1e-06$)
  - 16e-06 1.0e-6 1.8e-6 8.7e-4 1.3e-06 0.9e-06 . . . : $t_R \approx t_C \approx 1e-6$
  - 16e-06 1.1e-6 0.9e-6 1.0e-6 0.9e-6 1.1e-6 . . . : $t_R \ll t_C \approx 1e-6$
- note: a low t_R means a 'high (degree of) resolution'

Scales of Timings

- Whole applications
- Critical ‘inner loops’
  - how to identify these?
- Time for basic operations, eg. $+$, $*$
  - multiples of clock cycle
- Machine cycle time
  - 1GHz clock equivalent to 1nsec
  - note: cycle time is not always fixed!

Total Program Timing

C, Korn and Bourne shell provide the time and timex utility

```
me@gadi> time ./myprogram                      # This is under bash
real0m0.906s
user0m0.191s
sys0m0.688s
me@gadi> \time ./myprogram                     # actual comamand, e.g. /bin/time
0.17user 0.64system 0:00.83elapsed 97%CPU (0avgtext+0avgdata 728maxresident)k 0
    inputs+0outputs (0major+212minor)pagefaults 0swaps
me@gadi> \time -f "u=%Us s=%Ss e=%Es mem=%Mkb" ./cputime  # customize output
u=0.20s s=0.76s e=0:00.98es mem=732kb
```

- For parallel programs on multi-CPU machines, user time can exceed elapsed time
- High system time may indicate memory paging and/or I/O
- Ratio of user+system time to elapsed time can reflect other logged-in users
- we can customize output as indicated above

Manual Timing: Functions

```c
#include <stdio.h>
#include <sys/time.h>
#include <sys/times.h>
#include <time.h>
#include <unistd.h>
int main(int argc, char **argv) {
  struct tms cpu;
  struct timeval tp1, tp2;
  struct timezone tzp;
  gettimeofday(&tp1, NULL);
  long tick = sysconf(_SC_CLK_TCK);
  sleep(1);
  printf(" Ticks per second %ld \n", tick);
  gettimeofday(&tp2, NULL);
  times(&cpu);
  printf(" User   ticks %d \n", cpu.tms_utime);
  printf(" System ticks %d \n", cpu.tms_stime);
  printf(" Elapsed secs %d usec %d \n", tp2.tv_sec - tp1.tv_sec,
  tp2.tv_usec - tp1.tv_usec);
}
```

Manual Timing: Issues

- Resolution (and overhead): You should have some idea of its value
  - In some cases it may not be what is reported in a man page, e.g. it may say microseconds (1e-6) but are all the digits meaningful?
  - Often the resolution of the CPU timer is relatively low - one hundredth of a second is common
- CPU Time: Take care with the meaning of CPU time. Some timing functions switch from CPU to elapsed time if the program is running in parallel
- Baseline: Timing provides a baseline from which to judge performance tuning or comparative machine performance
- Placement: How do we know where to place timing calls!
  - Unix provides a number of profiling tools to help with this, e.g. prof, oprofile, etc
  - Other commercial offerings include VTune, Windows Performance Analysis Toolkit etc.

Performance Modeling

- Accurate performance models are needed to understand / predict performance
- Given a problem size n, typically the execution time is $t(n) = O(n^2)$
  - challenge generally in large n, not in complexity of $t(n)$
  - often (e.g. vector operations) $t(n) = a_0 + a_1 n$; the values of a_0, a_1 are important!
    i.e. $O(t(n))$ (tight upper bound), $\Omega(t(n))$ (lower), $\Theta(t(n))$ (upper+lower) concepts are inadequate
- A useful measure is the execution rate:

$$R(n) = \frac{g(n)}{t(n)}$$

where $g(n)$ is the algorithm's 'operation count', $g(n) = \Theta(t(n))$

  - e.g. graph of $R(n) = \frac{n}{10+n}$
  - note: if $g(n) = cn$, $a_0 =$ the startup cost, $c/a_1 = R(\infty) =$ the asymptotic rate
  - startup costs can be large, especially on vector computers
  - can use regression to determine a_0, a_1 by measuring $t(0), t(1000), \ldots$

Amdahl's Law #1

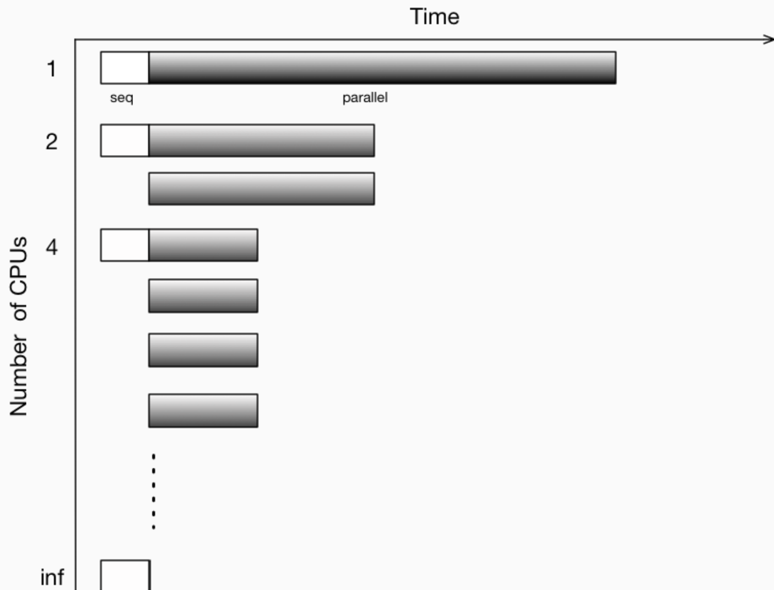
- The bane of parallel (||) HPC?
- Given a fraction f of 'slow' computation, at rate R_{s}, and R_{f} being the 'fast' computation rate:

$$R = (\frac{f}{R_{\mathrm{s}}} + \frac{1-f}{R_{\mathrm{f}}})^{-1}$$

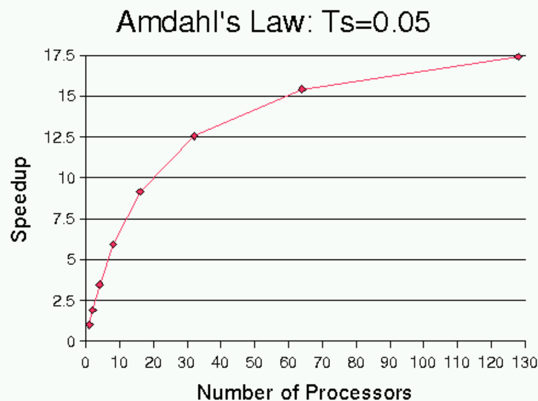
- Interpreted for vector processing:
  - f is the fraction of unvectorizable computation, with R_{f} (R_{s}) being the vector unit (scalar unit) speed
- Interpreted for parallel execution with p processors:
  - f is the fraction of serial computation, with $R_{\mathrm{f}} = pR_{\mathrm{s}}$, i.e.:

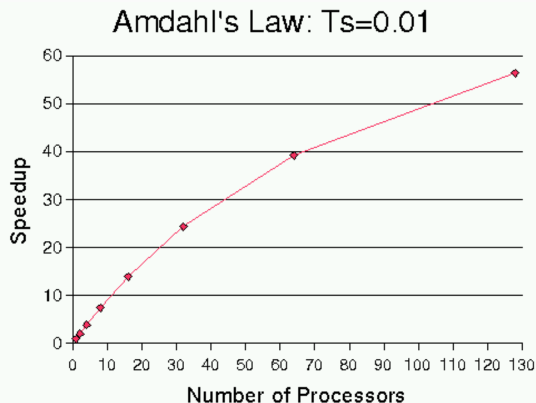
$$R_p = (f + \frac{1-f}{p})^{-1} R_{\mathrm{s}}$$

Amdahl's Law #2: Speedup



Amdahl's Law #3: Speedup Curves





"Better to have two strong oxen pulling your plough across the country than a thousand chickens. Chickens are OK, but we can't make them work together yet"

Amdahl's Law #4

- Other useful measures:
  - Speedup: $S_p = \frac{t_1}{t_p}$
    t_1 for the fastest serial algorithm, t_p is $||$ execution time
  - Efficiency: $E_p = \frac{S_p}{p}$
    ideally $E_p = 1$; is $E_p > 1$ possible?
- Consequences:
  - for a given fixed f, there will be a limit to p that can be usefully applied, eg. $p \leq \frac{1}{f}$
  - this set back $||$ computing 15 years!
- Counter notion: scalability
  - for a large p, only makes sense to use large n, ie. $n = n_1 p$
  - typically $f(n) = c'/n$, hence:
    $R(n) = R(n_1 p) = (\frac{c'}{n_1 p} + \frac{1 - c'/(n_1\ p)}{p})^{-1} R_{\mathrm{s}} \approx \frac{p}{c'/n_1 + 1} R_{\mathrm{s}}$
  - ie. $R(p)$ can increase linearly with p under these conditions
    $\Rightarrow ||$ processing can be worthwhile!

Hands-on Exercise: Timing and Computational Scaling

Objective:

- Check that your accounts are working
- Run some timing experiments to determine resolution and overhead

Outline

Introduction

Performance Measurement and Modeling

Example Applications

- Matrix Multiplication
- Heat-Stencil

Hardware Performance Counters

High Performance Microprocessors

Loop Optimization: Software Pipelining

Case Study: Matrix Multiplication

If **A** is a n x m matrix and **B** is a m x p matrix, their product **C** is a n x p matrix

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

where each entry C_{ij} is given by multiplying the entries $\mathbf{A}_{ik}$ (across row i of **A**) by the entries $\mathbf{B}_{kj}$ (down column j of **B**), for $k = 1, 2, ..., m$, and summing the results over k:

$$\mathbf{C}_{ij} = (\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

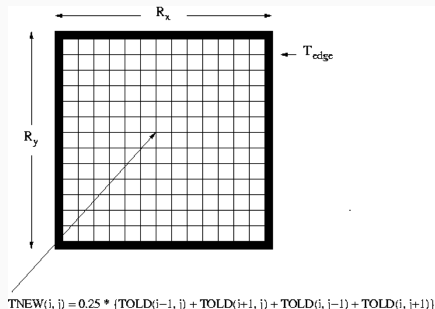
Source: https://en.wikipedia.org/wiki/Matrix_multiplication

Case Study: Heat-Stencil

- Stencil codes are iterative kernels which update array elements based on neighbouring values
- Two-dimensional heat diffusion is modelled by the Heat Equation

$$\frac{\partial u(t, \overrightarrow{x})}{\partial t} = \alpha \nabla^2 u(t, \overrightarrow{x})$$

- Example: given a metal plate of size R_x by R_y, where temperature at edges is held at T_{edge}, determine temperature at middle of the plate



  - The domain is divided into a grid of points. The new temperature for each grid point is calculated as the average of the current temperatures at the four adjacent grid points i.e. $TNEW(i,j) = \frac{TOLD(i-1,j)+TOLD(i+1,j)+TOLD(i,j-1)+TOLD(i,j+1)}{4}$
  - Iteration continues until the maximum change in temperature for any grid point is less than some threshold

Outline

Introduction

Performance Measurement and Modeling

Example Applications

Hardware Performance Counters

- PAPI

High Performance Microprocessors

Loop Optimization: Software Pipelining

Why Measure?

- Modern machines are complex, including:
    - pipelining
    - superscalar
    - load/store architectures
    - memory hierarchy
- Understanding observed performance is not easy
- Performance counters count critical events and provide an accurate means of assessing how well the computer system is being used

Hardware Performance Counters

- (Nearly?) All modern microprocessors have them
- Typically a group of registers that keep track of programmable events
- Provide high resolution data on many performance related variables, e.g.
  - cycles
  - instruction count
  - floating point operations
  - cache references
  - TLB misses
  - data/instruction stalls
- Enable vendors to better understand the performance of existing code on their hardware
- Enable users to build better (faster) software

Accessing Hardware Counters

- Cray YMP provided HPM that gave info on vector lengths and flops
    - enabled users to quote macho flops!
- Used to build metrics
- Used in system wide tools (e.g.: perf, gprof, tau, cputrack, cpustat etc.)
- Accessed via libraries
    - vendor specific (libcpc, libpctx, perflib)
    - portable (PCL, PAPI)
    - further GUI often provided for higher level analysis
- There are no standard counters
    - different vendors have different counters
    - different generations of the same processor may have different counters

Simple Metrics

$$\mathrm{MFLOPS} = \frac{FP_Instr_Exec}{Cycles} \times clock(MHz)$$

$$\mathrm{MIPS} = \frac{Int_Instr_Exec}{Cycles} \times clock(MHz)$$

$$\mathrm{IPC} = \frac{All_Instr_Exec}{Cycles}$$

$$\mathrm{L1Hits} = 1 - \frac{L1_Misses}{Loads + Stores}$$

$$\mathrm{L2Hits} = 1 - \frac{L2_Misses}{L1_Misses}$$

$$\mathrm{Branch_rate} = \frac{Decoded_Branches}{Total_Instruction_Exec}$$

More Complex Metrics

$$\mathrm{L1-L2_bandwidth} = \frac{L1_Misses \times L1_Line_Size}{Cycles} \times clock(MHz)$$

$$\mathrm{L2-RAM_bandwidth} = \frac{L2_Misses \times L2_Line_Size}{Cycles} \times clock(MHz)$$

$$\mathrm{Data_Stall} = \frac{Load_Use + Load_Use_Raw + Store_Buf_Full}{Cycles}$$

Intel Xeon (Sandy Bridge) HW Counters

- 11 hardware Performance Monitoring Units (48-bit wide)
  - 3 Fixed-function counters (FIXED_CTR0-FIXED_CTR2)
    - Each of these can count only one event
  - 8 General-purpose counters (PMC0-PMC7)
    - Each counter paired with a performance event select register PERFEVTSELx
    - Configure performance events via UMASK (unit mask) and the EVENT SELECT fields in the PERFEVTSELx
- i7 family is similar; 12 counters in total for Cascade Lake

Details for specific Intel processors at https://download.01.org/perfmon/index/

ARM Cortex-A8 Performance Counters

- 4 Performance Monitor CouNT Registers (PMCNT0-PMCNT3)
  - 32 bit counter
  - Each of PMCNT0-PMCNT3 registers selected by the PMNXSEL Register
- Event to be counted selected by the EVTSEL Register
- Performance Monitor Control (PMNC) Register controls the operation of the four Performance Monitor Count Registers

ARM64 Neoverse counters are also 32-bit but have 6 PMEVCNT registers, each controlled by a corresponding PMEVTYPE register

PAPI (https://icl.utk.edu/papi/)

- Portable library which provides a programming interface for the performance counter hardware
- Runs on most modern processors and operating systems
  - IBM POWER / AIX / Linux
  - Intel Pentium, Core2, Nehalem, SandyBridge, Cascade Lake / Linux
  - ARM Cortex, ARM64
- Countable events are defined in two ways:
  - Platform-neutral preset events (`papiStdEventDefs.h`): cache and branch events, cycle and instruction counts, functional units, pipeline status
  - Platform-dependent native events
- Presets can be derived from multiple native events

What PAPI provides

- Tools which provide information on hardware counters. e.g.
  - papi_avail
  - papi_cost
  - papi_mem_info
- High Level interface
  - Functions for coarse-grained measurements
- Low Level interface
  - Fine-grained measurements
  - Increased functionality

papi_cost

computes the cost of basic PAPI operations:

```
gadi:~$ papi_cost

Total cost for loop latency over 1000000 iterations
min cycles    : 18
max cycles    : 43812
mean cycles   : 28.638972
std deviation: 91.725368

Performing start/stop test...

Total cost for PAPI_start/stop (2 counters) over 1000000 iterations
min cycles    : 6200
max cycles    : 308616
mean cycles   : 7274.749000
std deviation: 1153.335188

Performing read test...

Total cost for PAPI_read (2 counters) over 1000000 iterations
min cycles    : 78
max cycles    : 44684
mean cycles   : 87.388440
std deviation: 148.663895

...
```

papi_avail

reports processor info and available present events:

```
gadi:~$ papi_avail
Available PAPI preset and user defined events plus hardware information.
--------------------------------------------------------------------------------
PAPI version              : 5.7.0.0
Operating system          : Linux 4.18.0-80.11.2.el8_0.x86_64
Vendor string and code    : GenuineIntel (1, 0x1)
Model string and code     : Intel(R) Xeon(R) Platinum 8268 CPU @ 2.90GHz (85, 0
    x55)
CPU revision              : 7.000000
CPUID                     : Family/Model/Stepping 6/85/7, 0x06/0x55/0x07
CPU Max MHz               : 3900
CPU Min MHz               : 1200
Total cores               : 48
SMT threads per core      : 1
Cores per socket          : 24
Sockets                   : 2
Cores per NUMA region     : 12
NUMA regions              : 4
Running in a VM           : no
Number Hardware Counters : 10
Max Multiplex Counters    : 384
PAPI_L1_DCM  0x80000000  Yes   No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  Yes   No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  Yes   Yes  Level 2 data cache misses
PAPI_L2_ICM  0x80000003  Yes   No   Level 2 instruction cache misses
...
```

papi_native_avail

reports available native events:

```
nc02202:~$ papi_native_avail
...
===============================================================================
 Native Events in Component: perf_event
===============================================================================
| ix86arch::UNHALTED_CORE_CYCLES                                               |
|            count core clock cycles whenever the clock signal on the specific |
|            core is running (not halted)                                      |
|     :e=0                                                                     |
|            edge level (may require counter-mask >= 1)                        |
|
--------------------------------------------------------------------------------
| ix86arch::INSTRUCTION_RETIRED                                                |
|            count the number of instructions at retirement. For instructions t|
|            hat consists of multiple micro-ops, this event counts the retireme|
|            nt of the last micro-op of the instruction                        |
...
```

PAPI High Level Interface

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (less additional code) than low-level
- Supports 8 calls in C or Fortran:

```
PAPI_start_counters        PAPI_stop_counters
PAPI_read_counters         PAPI_accum_counters
PAPI_num_counters          PAPI_flips
PAPI_ipc                   PAPI_flops
```

PAPI High Level Interface Example

```c
#include "papi.h"
#define NUM_EVENTS 2

long_long values[NUM_EVENTS];
unsigned int Events[NUM_EVENTS]={PAPI_TOT_INS,PAPI_TOT_CYC};

/* Start the counters */
PAPI_start_counters((int*)Events,NUM_EVENTS);

/* The workload to be monitored */
do_work();

/* Stop counters and store results in values */
retval = PAPI_stop_counters(values,NUM_EVENTS);
```

PAPI Low Level Interface

- Increased efficiency and functionality over the high level PAPI interface
- Obtain information about the executable, the hardware, and the memory environment
- Manages hardware events in user-defined groups called Event Sets.
- Allows both PAPI preset and native events unlike the High Level interface
- Multiplexing, Callbacks on counter overflow
- About 60 functions

PAPI Low Level Interface Example

```
#include "papi.h"
#define NUM_EVENTS 2
int Events[NUM_EVENTS]={PAPI_FP_INS,PAPI_TOT_CYC};
int EventSet;
long_long values[NUM_EVENTS];

/* Initialize the library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
/* Allocate space for the new eventset and do setup */
retval = PAPI_create_eventset(&EventSet);
/* Add FLOPs and total cycles to the eventset */
retval = PAPI_add_events(EventSet, Events, NUM_EVENTS);
/* Start the counters */
retval = PAPI_start(EventSet);

/* The workload to be monitored */
do_work();

/* Stop counters and store results in values */
retval = PAPI_stop(EventSet, values);
```

Hardware Performance Counters: Caveats

- Non-determinism due to hardware interrupts or external sources (e.g. OS interaction, program layout)
- Overcounting due to exceptions, microcode, context switching

V.M. Weaver, D. Terpstra, S. Moore (2013). *Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations*. ISPASS 2013

Hands-on Exercise: Hardware Performance Counters

Objective:

- Using PAPI to measure code performance

Outline

Introduction

Performance Measurement and Modeling

Example Applications

Hardware Performance Counters

High Performance Microprocessors

Loop Optimization: Software Pipelining

Instruction Set Architectures

- Early microprocessors were very simple, but in 1964 IBM introduced the 360 series which was micro-programmed
- From then instruction sets and addressing modes increased, prompted in part by development of high level languages
- Special microcode was added to handle case statements, procedure calling, array indexing etc
    - led to the CISC concept (Complex Instruction Set Computer)
- In the 70s writing, debugging and maintaining microcode became a major issue
- Academics begin to analyse what programs actually did and this resulted in a major rethink of microprocessor design
    - led to the RISC concept (Reduced Instruction Set Computer)

Characteristics of RISC and CISC Machines

	RISC	CISC
1	Simple instructions taking 1 cycle	Complex instructions taking multiple cycles
2	Only LOADS/STORES reference memory	Any instruction may reference memory
3	Highly pipelined	Not pipelined or less pipelined
4	Instructions executed by the hardware	Instructions interpreted by the microcode
5	Fixed format instructions	Variable format instructions
6	Few instructions and modes	Many instructions and modes
7	Complexity is in the compiler	Complexity is in the micro-program
8	Multiple register sets	Single register set

Assembly language programmers used the complicated machine instructions, but compilers generally did not. Difficult to get compiler to recognize complicated instructions.

RISC is now the dominant scalar processor architecture

RISC Processors

First Generation Characteristics

- Pipelining (both instruction and floating point)
- Branching (delayed branching and branch prediction)
- Uniform instruction length
- Load/Store architecture (simple addressing)

Second Generation

- Faster clocks
- Super-pipelining
- Superscalar

Post-RISC

- Out-of-order execution

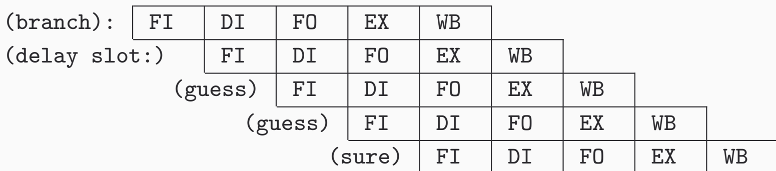
Pipelining

Everything happens in step with the clock. Overlap instructions so that more than one can be in progress at any time. RISC architectures can initiate an instruction each cycle, but previous instructions may not have completed.

- Break instr'n execution into k stages; $\Rightarrow$ can get $\leq k$-way $||$ism

  (generally, the circuitry for each stage is independent)

- e.g. $(k = 5)$: stages FI = Fetch Instrn., DI = Decode Instrn., FO = Fetch Operand, EX = Execute Instrn., WB = Write Back



  - note: the FO & WB stages may involve memory accesses (and may possibly stall the pipeline)

Pipelining: Dependent Instructions

CPU must ensure result is the same as if no pipelining (or ||ism)

- Instructions requiring only 1 cycle in the EX stage:

```
add %1, -1, %1        ! r1 = r1 - 1    (integer register subtract)
cmp %1, 0             ! is r1 = 0?     (integer register compare)
```

  can be solved by pipeline feedback from EX stage to next cycle

- (important) Instr'ns requiring $c > 1$ cycles for the EX stage (e.g. f.p. * +, load, store) are normally implemented by having c EX stages, delaying dependent instr'n by c cycles e.g. $c = 3$

```
fmuld %f0, %f2, %f4 ! I0: fr4 = fr0 * fr2 (f.p. register multiply)
....                ! I1:
....                ! I2:
faddd %f4, %f6, %f6 ! I3: fr6 = fr4 + fr6 (f.p. register add)
```

I0:	FI	DI	FO	EX1	EX2	EX3	WB			
I1:		FI	DI	FO	EX1	EX2	EX3	WB		
I2:			FI	DI	FO	EX1	EX2	EX3	WB	
I3:				FI	DI	FO	EX1	EX2	EX3	WB

Pipelining: Dependent Instructions(cont)

Notes:

- If I3 is $\delta < c$ cycles after I0, the CPU must insert $c - \delta$ pipeline bubbles (NoOps) in between.
  Can avoid this by software pipelining: (where possible) separate I3 from I0 in the original code by at least c cycles
- EX2, EX3 may be 'empty' for the simpler instructions (e.g. int $+$)
- Less important instrn.'s requiring larger c (e.g. f.p. $/$, int $*$, $/$, %) are either not pipelined or use a separate sub-pipeline for their EX stages

Pipelining: Branch Instructions

A branch to a new program address (perhaps caused by an if statement) will disrupt the pipeline flow. The processor doesn't know if the instruction is a branch until the decode stage and then may not know if it will be taken until the execute stage. If the branch is taken then following "in flight" instructions must be annulled.

- Many processors require a 'branch delay slot' instruction immediately after the branch instruction. This enables the pipeline to continue for **unconditional branches** (ie when the decoded instructions says branch somewhere and we go there). Conditional branches are more difficult, pipeline will stall and require flushing as it can't be recognized before the DI stage

  e.g.

```
cmp %1, %2           ! n = n + 1
bne endif1           ! if (i == k) ...
add %3,1,%3          ! delay slot - ALWAYS executed
```

(if possible try to move a logically preceding instr'n into the delay slot)

Pipelining: Branch Prediction

- To handle conditional branches various branch prediction schemes are used:
  - Assume branches are always taken (flush pipeline when not taken) (OK for loops, with test at bottom)
  - S/W (compiler) indicates the 'most likely' prediction
  - H/W keeps a branch prediction buffer: predict using the result of the last (few) executions of the branch (2 bit common)

Pipelines and Floating-Point Operations: Summary

- FP operations typically take longer than fixed point operations so benefit greatly from pipelining.
  - The number of stages in the pipeline may be increased so even complicated operations like FP * can be pipelined.
- FP +, -, *, comparison and conversion are pipelined
  - Usually sqrt and / are NOT pipelined
- Some processors limit overlap of FP operations due to shared internal components
  - Fully pipelined $\Rightarrow$ no overlap restrictions

Load/Store Architecture

- Memory reference are restricted to load/store
  - Only one reference per instruction
  - In CISC, arithmetic/logical instructions may include a memory reference
- Motivation:
  - To enable fixed instruction length
  - To ease pipelining
  - Since memory references may be slow

Second Generation RISC Processors

After proving basic concept

- Improvement in manufacturing led to faster clock rates
- Increase pipeline stages making each stage simpler and faster
- Add multiple compute elements: Superscalar

Superscalar (multiple instruction issue)

A small number (w) of instructions are scheduled by the H/W to execute together

- Groups must have an appropriate 'instruction mix'

  eg. UltraSPARC ($w = 4$): $\left\{ \begin{array}{l} \leq 2 \text{ }_{\text{different}} \text{ floating point} \\ \leq 1 \text{ load / store ; } \leq 1 \text{ branch} \\ \leq 2 \text{ integer / logical} \end{array} \right\}$ instructions per group
- Have $\leq w$-way $||$ism over different types of instructions
- Generally requires:
  - Multiple ($\geq w$) instruction fetches
  - Extra grouping (G) stage in the pipeline
- Problem: will require a deeper software pipelining (by a factor of w)
  - Generally, all problems with pipelining are similarly amplified
- Issues: the instruction mix must be balanced for maximum performance!
  - NB. floating point $*$, $+$ must be balanced

Post-RISC Architecture

- Two-way superscalar successful and in 1994 able to run at 1.6-1.8 instructions per cycle
- "Higher-way" superscalar may appear natural progression, but difficult to find enough instruction level parallelism to justify
- Speculative execution or out-of-order execution is more popular. Permits instructions to be executed that may never be used, e.g. in the following FDIV may be elevated up the execution stack if sufficient space is present to store the result

```
LD     R10,R2(r0)     Load into R10 from memory
.
.                     many instructions of various kinds but no FDIV
.
FDIV   R4,R5,R6       R4 = R5 /R6
```

- Out-of-order processors include a *instruction reorder buffer* to store instructions that are in limbo

In-order vs. Out-of-order Execution

- In-order instruction execution
    - Instructions are fetched, executed & completed in compiler-generated order
    - One stalls, they all stall
    - Instructions are statically scheduled
- Out-of-order instruction execution
    - Instructions are fetched in compiler-generated order
    - Instruction completion may be in-order (today) or out-of-order (older computers)
    - In-between, they may be executed in some other order
    - Independent instructions behind a stalled instruction can pass it
    - Instructions are dynamically scheduled

Dynamic Scheduling

Out-of-order processors:

- After instruction decode
    - Check for structural hazards
        - An instruction can be issued when a functional unit is available
        - An instruction stalls if no appropriate functional unit
    - Check for data hazards
        - An instruction can execute when its operands have been calculated or loaded from memory
        - An instruction stalls if its operands are not available

Summary

- RISC is now the dominant architecture type
    - Modern x86 processors mix elements of CISC and RISC
- Typical pipelines are 5-15 stages and instructions are 3-4 way superscalar
- Can only achieve up to inherent parallelism in instruction stream
- Dependent instructions must be sufficiently separated by either:
    1. S/W (need good compilers & large # registers)
    2. H/W (if done via dynamic instruction reordering, this is more effective, but harder to achieve!)

Hands-on Exercise: Pipelining

Objective:

- to interpret assembly language and understand dependencies between instructions

Outline

Introduction

Performance Measurement and Modeling

Example Applications

Hardware Performance Counters

High Performance Microprocessors

Loop Optimization: Software Pipelining

Loop Unrolling and Software Pipelining#1

- Consider the loop

```
for (i = 0; i < N; i++) {
  y[i] = y[i] + a * x[i]
}
```

- Running on a system with:
  - Load/store latency of 2 cycles to L1 cache
  - fmul/fadd latency of 3 cycles (EX stages)
  - Superscalar with 1 ld/st, 2 FP, 2 Int ops
- How many cycles to execute 1 loop iteration?

Loop Unrolling and Software Pipelining#2

```
                                                  ! Instruction Groups
for (i = 0; i < N; i++) { | for(i=0;i<N;i++){ ! Issue                  Completes
  y[i] = y[i] + a * x[i]; |   x0 = x[i]        ! [1] ld(x0)             //repeat[10]
}                         |   y0 = y[i]        ! [2] ld(y0)             ..st(y0)
                          |   x0 = x0 * a      ! [3] fmul(x0,a)         ld(x0)
                          |                    ! [4] -                  ld(y0)
                          |                    ! [5] -
                          |   y0 = y0 + x0     ! [6] fadd(x0,y0)        fmul(x0,a)
                          |                    ! [7] -
                          |                    ! [8] -
                          |   y[i] = y0        ! [9] st(y0),blt(i,n)    fadd(x0,y0)
                          | }
```

- Loop takes 9 cycles to complete 1 iteration
  - In 4 cycles no instructions are issued!
  - Only once do we use the superscalar capabilities (st(y0),blt(i,n))

Loop Unrolling and Software Pipelining#3

- What if we "unroll" the loop by a factor of 2?

```
for (i = 0; i < N%2; i++){             ! preconditioning loop
  y[i] = y[i] + a * x[i];
}
for (i = N%2; i < N; i+=2){
  y[i]   = y[i]   + a * x[i];
  y[i+1] = y[i+1] + a * x[i+1];
}
```

- Reduces loop overhead
- Exposes more possibilities for instruction ||ism
  - We can software pipeline the operations

Loop Unrolling and Software Pipelining#4

```
                                   ! Instruction Groups
for (i = N%2; i < N; i+=2) {  ! Issue                       Completes
  x0 = x[i];                    !  [1] ld(x0)                 ..st(y0)//Repeat[11]
  x1 = x[i+1];                  !  [2] ld(x1)                 ..st(y1)
  y0 = y[i];    x0 = x0 * a;    !  [3] ld(y0),fmul(x0,a)      ld(x0)
  y1 = y[i+1]; x1 = x1 * a;     !  [4] ld(y1),fmul(x1,a)      ld(x1)
                                !  [5] -                      ld(y0)
  y0 = y0 + x0;                 !  [6] fadd(x0,y0)            ld(y1),fmul(x0,a)
  y1 = y1 + x1;                 !  [7] fadd(x1,y1)            fmul(x1,a)
                                !  [8] -
                                !  [9] st(y0)                 fadd(x0,y0)
}                               ! [10] st(y1),blt(i,n)        fadd(x1,y1)
```

- Now obtain 2 results every 10 cycles, or effectively 1 result every 5 cycles
- Further unrolling will give 1 result every 3 cycles, i.e.
    - ultimately the loop is load/store dominated.
- Note: poor instruction mix at the start of the loop

Loop Unrolling and Software Pipelining#5

- Greater unrolling also permits better hiding of L2 cache load/store latencies (e.g. delay of 8 cycles instead of 2!)
- With moderate levels of optimization (e.g. -O3), compilers generally unroll inner loops automatically. But you may need to look at the assembler code to see exactly what is done.
- In general unrolling is inadvisable when loop:
  - has a low trip count: ie. N is small
    - because extra setup is needed
  - body is already fat $\Rightarrow$ register spilling
    - generally, the unrolling should match (the register level of) the memory hierarchy
  - has (unavoidable) procedure calls

✗ note: unrolling increases code size

Dependencies and Aliasing#1

- Pointer aliasing

```
void vadd(int n, double a[], double b[]) {
  int k;
  for (k = 0; k < n; k++) {
     a[k] = a[k] + b[k];
  }
}
```

- Consider:
  - `vadd(n, &a[0], &a[0]);  // a[i] = a[i] + a[i]`
  - `vadd(n, &a[0], &a[1]);  // a[i] = a[i] + a[i+1]`
  - `vadd(n, &a[1], &a[0]);  // a[i+1] = a[i+1] + a[i]`
  - `vadd(n, &a[0], &a[n]);  // a[i] = a[i] + a[i+n]`

Dependencies and Aliasing#2

- What if loop unrolling causes load of data for iteration i+1 to occur before store of data iteration i.

```
Iter i                      Iter i+1                    Iter i+2
------------------------------------------------------------------
ld a[i], r1
ld b[i], r2
fadd r1, r2, r3
                            ld a[i+1], r4
                            ld b[i+1], r5
st r3, a[i],                fadd r4, r5, r6
                                                        ld a[i+2], r7
                                                        ld b[i+2], r8
                            st r6, a[i+1]               fadd r7, r8, r9
```

- This will give the wrong result for `a[i+1] = a[i+1]+a[i]`
- By default C/C++ assumes pointers can be aliased. This limits pipelining, so moderate compiler optimization removes this restriction ...
  - with the potential of wrong results!

Loops with Inter-Iteration Dependencies: Reductions

- Hard to extract any instruction parallelism at all!
- e.g. 'scan' an array:

```
y[1] = 0.0
for(i = 2; i < N; i++) {
  y[i+1] = y[i] + x[i];
}
```

- Reductions: special case of scan algorithm: inter-iteration dependencies are over a scalar variable
- e.g. inner product of 2 vectors

```
s = 0.0;                          | s = 0.0;
for (i = 0; i < N; i++) {         | for (i=0;i<N;i++) {  ! Cycle
  s = s + x[i] * y[i];            |   x0 = x[i]           ! [1] //repeat [10]
}                                 |   y0 = y[i]           ! [2]
                                  |   x0 = x0 * y0        ! [4] wait ld(y0)
                                  |   s = s + x0          ! [7] wait mult(x0,y0)
                                  |                       !     wait add(s,x0)
                                  | }
```

Loop with Inter-Iteration Dependencies: Reductions#2

- Unrolling inner product by 2:

```
  ...
for (i = N%2; i < N; i+=2) {              ! Cycle
  x0 = x[i]                               ! [1]      //repeat [13]
  x1 = x[i+1]                             ! [2]      //add(s,x1) completes
  y0 = y[i]                               ! [3]
  y1 = y[i+1]                             ! [4]
              x0 = x0 * y0                ! [5]
              x1 = x1 * y1                ! [6]
              s = s + x0                  ! [8]  wait mult(x0,y0)
              s = s + x1                  ! [11] wait add(s,x0)
                                          !      wait add(s,x1)
}                                         ! loop book-keeping overlaps
```

- 12 cycles for 2 results (compared to 9 cycles for 1 iteration)
- How can performance be further improved?
  - Change order at start of loop
  - Remove dependencies of += op'ns

Loop with Inter-Iteration Dependencies: Reductions#3

```
s1=0; s2=0;
for(i = N%2; i < N; i+=2){                      ! Cycle
  x0 = x[i]                                     ! [1]  //repeat [10] wait mult(x0,y0)
  y0 = y[i]                                     ! [2]      <<NEW order
  x1 = x[i+1]                                   ! [3]
  y1 = y[i+1]; x0 = x0 * y0                     ! [4]      <<OVERLAP
               x1 = x1 * y1                     ! [5]
               s1 = s1 + x0                     ! [7]  wait mult(x0,y0)
               s2 = s2 + x1                     ! [8]  wait mult(x1,y1)
}                                               ! loop book-keeping overlaps
s = s1 + s2
```

- 9 cycles for 2 results (cf. 12 cycles for 2 results)

Hands-on Exercise: Loop Ordering and Unrolling

Objective:

- To investigate the effect of loop ordering and loop unrolling on performance
- To see compiler generated loop unrolling in the assembly code and to understand what is meant by aliasing and its implications

Summary

Topics covered today - CPU Architecture:

- Performance measurement and modeling
- Hardware performance counters
- Key features of modern processors
- Loop optimization

Tomorrow - Vectorization & Cache Organization!