Modern Processors & Hardware Support for Performance Measurement

Dr. John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@cs.rice.edu

Motivating Questions

- What does good performance mean?
- How can we tell if a program has good performance?
- How can we tell that it doesn’t?
- If performance is not “good,” how can we pinpoint where?
- How can we identify the causes?
- What can we do about it?

Application Performance

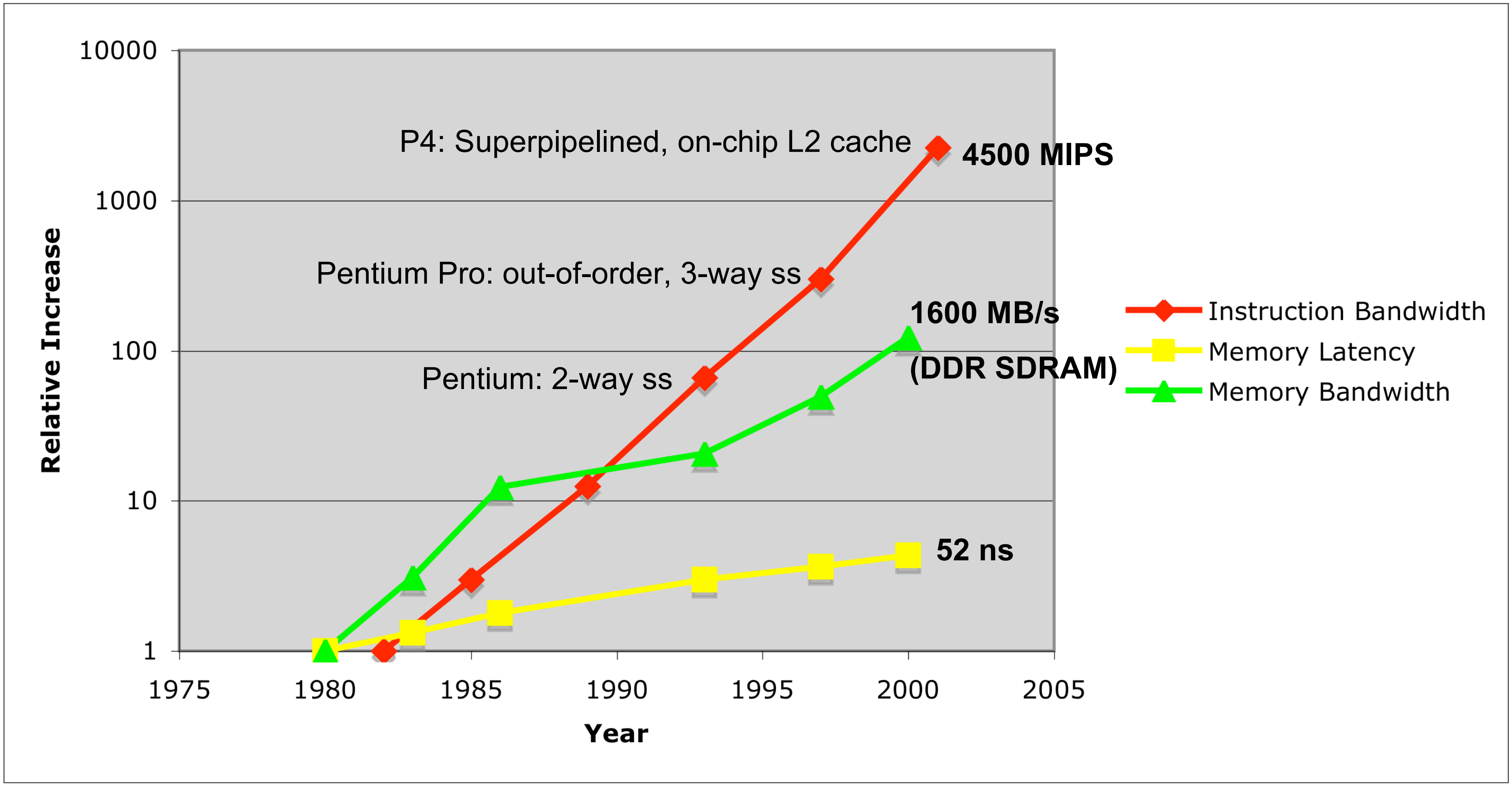
- **Performance is an interaction between**
  - —Numerical model
  - —Algorithms
  - —Problem formulation (as a program)
  - —Data structures
  - —System software
  - —Hardware
- **Removing performance bottlenecks may require dependent adjustments to all**

Goals for Today

Understand

- **Factors affecting performance on microprocessor architectures**
- **Organization of modern microprocessors**
- **Performance monitoring hardware capabilities**
  - —event counters
  - —instruction sampling
- **Strategies for measuring application node performance**
  - —Performance calipers
  - —Sample-based profiling
  - —How and when to use each

A Stable Trend: The Widening Gap to Memory



Data from

D. Patterson, *Latency Lags Bandwidth*, CACM 47(10), Oct. 2004.

Peak vs. Realized Performance

Peak performance = guaranteed not to exceed
Realized performance = what you achieve

Scientific applications realize as low as 5-25% of peak on microprocessor-based systems

Reason: mismatch between application and architecture capabilities

- —Architecture has insufficient bandwidth to main memory:
  - – microprocessors often provide < 1 byte from memory per FLOP
    - • scientific applications often need more
- —Application has insufficient locality
  - – irregular accesses can squander memory bandwidth
    - • use only part of each data block fetched from memory
  - – may not adequately reuse costly virtual memory address translations
- —Exposed memory latency
  - – architecture: inadequate memory parallelism to hide latency
  - – application: not structured to exploit memory parallelism
- —Instruction mix doesn't match available functional units

Performance Analysis and Tuning

- Increasingly necessary
  - —Gap between realized and peak performance is growing
- Increasingly hard
  - —Complex architectures are harder to program effectively
    - – complex processors: pipelining, out-of-order execution, VLIW
    - – complex memory hierarchy: multi-level non-blocking caches, TLB
  - —Optimizing compilers are critical to achieving high performance
    - – small program changes may have a large impact
  - —Modern scientific applications pose challenges for tools
    - – multi-lingual programs
    - – many source files
    - – complex build process
    - – external libraries in binary-only form

Performance = Resources = Time

$$T_{program} = T_{compute} + T_{wait}$$

$T_{compute}$ is the time the CPU thinks it is busy.

T_{wait} is the time it is waiting for external devices/events.

$$T_{compute} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Determined by model, algorithm, and data structures.

Determined by architecture and by the compiler's ability to use the architecture efficiently *for your program*.

Determined by technology and by hardware design

Including:

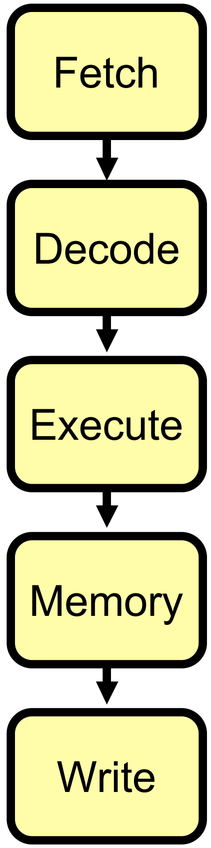
other processes, operating system, I/O, communication

Microprocessor-based Architectures

- Instruction Level Parallelism (ILP): systems not really serial
  - —Deeply pipelined processors
  - —Multiple functional units
- Processor taxonomy
  - —Out-of-order superscalar: Alpha, Pentium 4, Opteron
    - – hardware dynamically schedules instructions: determine dependences and dispatch instructions
    - – many instructions in flight at once; instructions execute out of order
  - —VLIW: Itanium
    - – issue a fixed size “bundle” of instructions each cycle
    - – bundles tailored to mix of available functional units
    - – compiler pre-determines what instructions initiate in parallel
- Complex memory hierarchy

Pipelining 101

- Basic microprocessor pipeline (RISC circa 1983)
  - Instruction fetch (IF)
  - Instruction decode (ID)
  - Execute (EX)
  - Memory access (MEM)
  - Writeback (WB)
- Each instruction takes 5 cycles (latency)
- One instruction can complete per cycle (theoretical peak throughput)
- Disruptions and replays
  - On simple processors: bubbles and stalls
  - Recent complex/dynamic processors use an abort/replay approach



Limits to Pipelining

- **Hazards: conditions that prevent the next instruction from being launched or (in speculative systems) completed**
  - **—Structural hazard: Can't use the same hardware to do two different things at the same time**
  - **— Data hazard: Instruction depends on result of prior instruction still in the pipeline. (Also, instruction tries to overwrite values still needed by other instructions.)**
  - **— Control hazard: Delay between fetching control instructions and decisions about changes in control flow (branches and jumps)**
- **In the presence of a hazard, introduce delays (pipeline bubbles) until the hazard is resolved**
- **Deep or complex pipelines increase the cost of hazards**
- **External Delays**
  - **— Cache and memory delays**
  - **— Address translation (TLB) misses**

Out-of-order Processors

- **Dynamically exploit instruction-level parallelism**
  - —fetch, issue multiple instructions at a time
  - —dispatch to several function units
  - —retire up to several instructions (maximum fixed) in a cycle
- **What are ordering constraints?**
  - —fetch in-order
  - —execute out of order
    - – map architectural registers to physical registers with renaming to avoid conflicts
    - – abort speculatively executed instructions (e.g. from mispredicted branches)
  - —retire in-order

Sources of Delay in Out-of-Order Processors

- **Fetcher may stall**
  - — icache miss (data hazard)
  - —mispredicted branch (control hazard)
- **Mapper may stall**
  - —lack of free physical registers (structural hazard)
  - —lack of issue queue slots (structural hazard)
- **Instructions in issue queue may stall**
  - —wait for register dependences to be satisfied (data hazard)
  - —wait for functional unit to be available (structural hazard)
- **Instruction execution may stall**
  - —waiting for data cache misses (data hazard)

Pentium 4 (Super-Pipelined, Super-Scalar)

- **Stages 1-2**
  - Trace cache next instruction pointer
- **Stages 3-4**
  - Trace cache fetch
- **Stage 5**
  - Drive (wire delay!)
- **Stages 6-8**
  - Allocate and Rename
- **Stages 10-12**
  - Schedule instructions
  - Memory/fast ALU/slow ALU & general FP/simple FP
- **Stages 13-14**
  - Dispatch
- **Stages 15-16**
  - Register access
- **Stage 17**
  - Execute
- **Stage 18**
  - Set flags
- **Stage 19**
  - Check branches
- **Stage 20**
  - Drive (more wire delay!)

5 operations issued per clock
1 load, 1store unit
2 simple/fast, 1 complex/slower integer units
1 FP execution unit, 1 FP move unit
Up to 126 instructions in flight: 48 loads, 24 stores, …

Opteron Pipeline (Super-Pipelined, Super-Scalar)

Integer

1. Fetch1
2. Fetch2
3. Pick
4. Decode1
5. Decode2
6. Pack
7. Pack/ Decode
8. Dispatch
9. Schedule
10. AGU/ALU
11. DC access
12. DC Response

Floating Point

8. Dispatch
9. Stack rename
10. Register rename
11. Sched. Write
12. Schedule
13. Reg. Read
14. FX0
15. FX1
16. FX2
17. FX3
48. …

L2 Cache

13. L2 Tag
14. …
15. L2 Data
16. …
17. Route/ Multiplex ECC
18. …
19. Write DC/ Forward Data

DRAM

14. Address to SAQ
15. Clock Crossing
16. Sys. Req Queue
17. …
26. Req DRAM pins
27. …

… (Memory Access)

Fetch/decode 3 inst/cycle.
3 integer units
3 address units
3 FPU/multimedia units
2 load/stores to d-cache/clk

Itanium2 Pipeline (VLIW/EPIC)

Front End

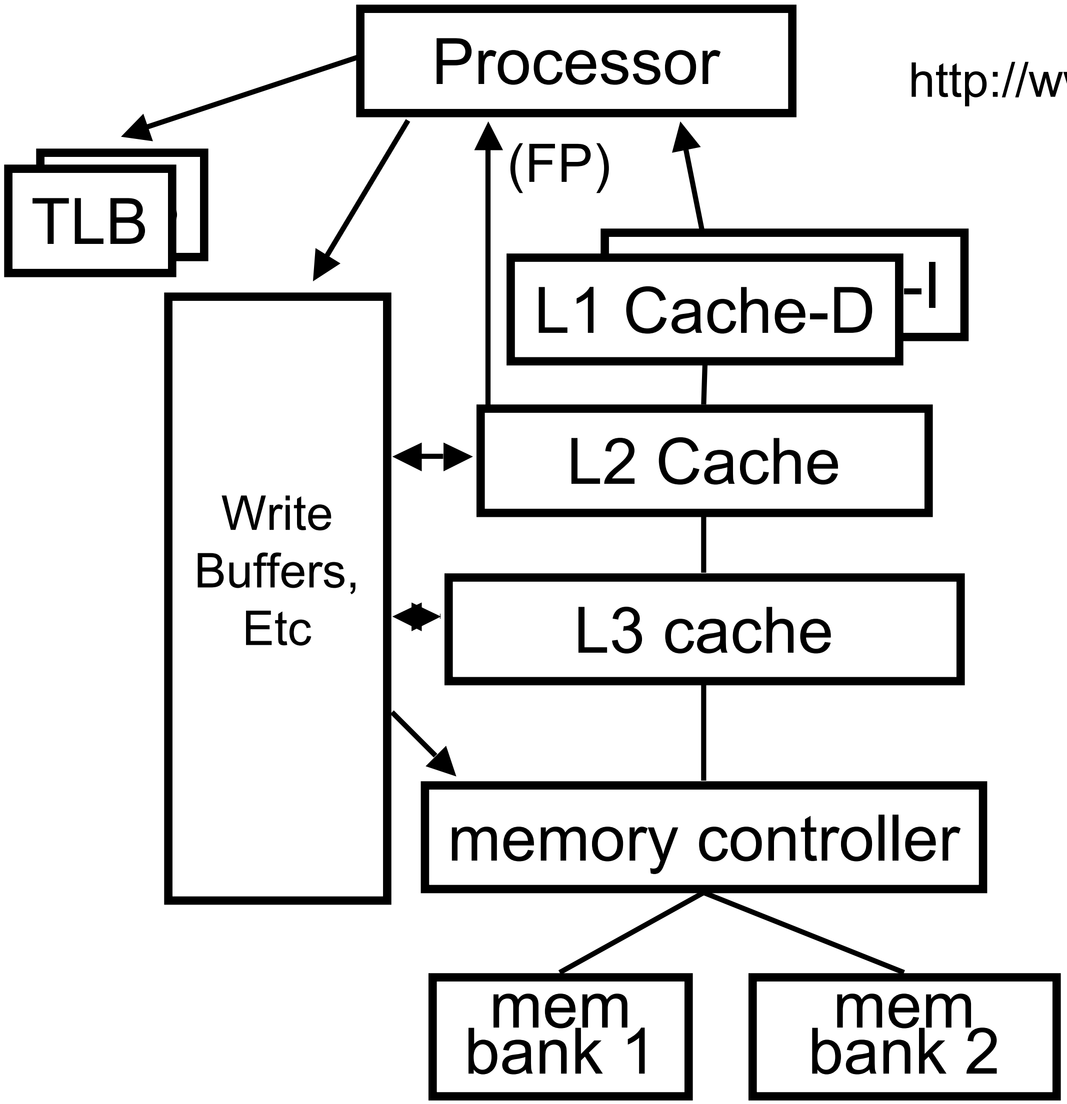
1. **Inst. Ptr. Generation and Fetch**
2. **Inst. "Rotation"**

Back End

3. **Decode, expand, and disperse**
4. **Rename and decode registers**
5. **Register file read**
6. **Execution/ Memory(Cache) 1/ FP1**
7. **Exception detection/ Memory(Cache) 2/ FP2**
8. **Write back/ Memory(Cache) 3/ FP3**
9. **Memory(Cache) 4/ FP4**

Six instructions (two bundles) issued per clock.
2 integer units
4 memory units
3 branch units
2 floating-point

A Modern Memory Hierarchy (Itanium 2)



http://www.devx.com/Intel/Article/20521

16K I + 16K D, 1 cycle

256K, 5 (6 FP) cycles

3M, 13.3 (13.1 FP cycles)

209.6 ns

Memory Hierarchy Components

- Translation Lookaside Buffer (TLB)
  - —Fast virtual memory map for a small (~64) number of pages.
  - —Touch lots of pages →lots of TLB misses →expensive
- Load/Store queues
  - —Write latency and bandwidth is usually* not an issue
- Caches
  - —Data in a cache is organized as set of 32-128 byte blocks
  - —Spatial locality: use all of the data in a block
  - —Temporal locality: reuse data at the same location
  - —Load/store operations access data in the level of cache closest to the processor in which data is resident
    - – load of data in L1 cache does not cause traffic between L1 & L2
  - —Typically, data in a cache close to the processor is also resident in lower level caches (inclusion)
  - —A miss in L_k cache causes an access to L_{k+1}
- Parallelism
  - —Multi-ported caches
  - —Multiple memory accesses in flight

The Memory Wall: LMBENCH Latency Results

Processor.	P4, 3.0GHz I865PERL	Dual K8, 1.6GHz Tyan2882	Dual K8, 1.6GHz Tyan2882	McKinley 900MHz x 2 HP zx6000	P4 2GHz Dell
Memory	PC3200	Registered PC2700ECC node IL on	Registered PC2700ECC node IL off	PC2100ECC	RB800
Compiler	gcc3.2.3	gcc3.3.2	gcc3.3.2	gcc3.2	gcc3.2
Integer +/*/Div (ns)	.17/4.7/19.4	.67/1.9/26	.67/1.9/26	1.12/4/7.9	.25/.25/11
Double +/*/Div (ns)	1.67/2.34/14.6	2.54/2.54/11.1	2.54/2.54/11.1	4.45/4.45	2.5/3.75/
Lat. L1 (ns)	0.67	1.88	1.88	2.27	1.18
Lat. L2 (ns)	6.15	12.40	12.40	7.00	9.23
Lat. Main (ns)	91.00	136.50	107.50	212.00	176.50

Note: 64 bytes/miss @ 100 ns/miss delivers only 640 MB/sec

The Memory Wall: Streams (Bandwidth) Benchmark

Processor.	P4, 3.0GHz I865PERL	Operon (x2), 1.6GHz Tyan2882	McKinley(x2), 900MHz HP zx6000
Bus/Memory	800MHz PC3200	Registered PC2700ECC node IL off	PC2100ECC
Native compiler 6M elements	icc8.0 -fast		ecc7.1 -O3
copy	2479		3318
scan	2479		3306
add	3029		3842
triad	3024		3844
gcc 6M elements	gcc3.2.3 -O3	gcc3.3.2 -O3 -m64	gcc3.2 -O3 (-funroll-all-loops)
copy	2422	1635	793 (820)
scan	2459	1661	734 (756)
add	2995	2350	843 (853)
triad	2954	1967	844 (858)

good!

terrible!

Itanium requires careful explicit code scheduling!

Take Home Points

- **Modern processors are complex and require instruction-level parallelism for performance**
  - **Understanding hazards is the key to understanding performance**
- **Memory is much slower than processors and a multi-layer memory hierarchy is there to hide that gap**
  - **The gap can't be hidden if your bandwidth needs are too great**
  - **Only data reuse will enable you to approach peak performance**
- **Performance tuning may require changing <u>everything</u>**
  - **Algorithms, data structures, program structure**

Survey of Hardware Performance Instrumentation

Performance Monitoring Hardware

Purpose

- Capture information about performance critical details that is otherwise inaccessible
  - cycles in flight, TLB misses, mispredicted branches, etc

What it does

- Characterize "events" and measure durations
- Record information about an instruction as it executes.

Two flavors of performance monitoring hardware

1. Aggregate performance event counters
   - sample events during execution: cycles, cache misses, etc.
   - limitation: out-of-order execution smears attribution of events
2. "ProfileMe" instruction execution trace hardware
   - a set of boolean flags indicating occurrence of events (e.g., traps, replays, etc) + cycle counters
   - limitation: not all sources of delay are counted, attribution is sometimes unintuitive

Types of Performance Events

- Program characterization
  - —Attributes independent of processor's implementation
    - – Number and types of instructions, e.g. load/store/branch/FLOP
- Memory hierarchy utilization
  - —Cache and TLB events
  - —Memory access concurrency
- Execution pipeline stalls
  - —Analyze instruction flow through the execution pipeline
  - —Identify hazards
    - – e.g. conflicts that prevent load/store reordering
- Branch Prediction
  - —Count mispredicts to identify hazards for pipeline stalls
- Resource Utilization
  - —Number of cycles spent using a floating point divider

Performance Monitoring Hardware

- Event detectors signal
  - —Raw events
  - —Qualified events, qualified by
    - – Hazard description
      - MOB_load_replay: +NO_STA, +NO_STD, +UNALGN_ADDR, …
    - – Type specifier
      - page_walk_type: +DTMISS, +ITMISS
    - – Cache response
      - ITLB_reference: +HIT, +MISS
    - – Cache line specific state
      - BSQ_cache_reference_RD: +HITS, +HITE, +HITM, +MISS
    - – Branch type
      - retired_mispred_branch: +COND, +CALL, +RET, +INDIR
    - – Floating point assist type
      - x87_assist: +FPSU, +FPSO, +POAU, +PREA
    - – …
- Event counters

Counting Processor Events

Three ways to count

- **Condition count**
  - —Number of cycles in which condition is true/false
    - – e.g. the number of cycles in which the pipeline was stalled
- **Condition edge count**
  - —Number of cycles in which condition changed
    - – e.g. the number of cycles in which a pipeline stall began
- **Thresholded count**
  - —Useful for events that report more than 1 count per cycle
    - – e.g. # cycles in which 3 or more instructions complete
    - – e.g. # cycles in which 3 or more loads are outstanding on the bus

Key Performance Counter Metrics

- **Cycles: PAPI_TOT_CYC**
- **Memory hierarchy**
  - **—TLB misses**
    - **PAPI_TLB_TL (Total), PAPI_TLB_DM (Data), PAPI_TLB_IM (Instructions)**
  - **—Cache misses:**
    - **PAPI_L*k*_TCM (Total), PAPI_L*k*_DCM (Data), PAPI_L*k*_ICM (Instructions)**
      - ***k* in [1 .. Number of cache levels]**
    - **Misses: PAPI_L*k*_LDM (Load), PAPI_L*k*_STM (Store)**
      - ***k* in [1 .. Number of cache levels]**
- **Pipeline stalls**
  - **PAPI_STL_ICY (No-issue cycles), PAPI_STL_CCY (No-completion cycles)**
  - **PAPI_RES_STL (Resource stall cycles), PAPI_FP_STAL (FP stall cycles)**
- **Branches: PAPI_BR_MSP (Mispredicted branch)**
- **Instructions: PAPI_TOT_INS (Completed), PAPI_TOT_IIS (Issued)**
  - **—Loads and stores: PAPI_LD_INS (Load), PAPI_SR_INS (Store)**
  - **—Floating point operations: PAPI_FP_OPS**
- **Events for shared-memory parallel codes**
  - **—PAPI_CA_SNP (Snoop request), PAPI_CA_INV (Invalidation)**

Useful Derived Metrics

- Processor utilization for this process
  - cycles/(wall clock time)
- Memory operations
  - load count + store count
- Instructions per memory operation
  - (graduated instructions)/(load count + store count)
- Avg number of loads per load miss (analogous metric for stores)
  - (load count)/(load miss count)
- Avg number of memory operations per L_k miss
  - (load count + store count)/(L_k load miss count + L_k store miss count)
- L_k cache miss rate
  - (L_k load miss count + L_k store miss count)/(load count + store count)
- Branch mispredict percentage
  - 100 * (branch mispredictions)/(total branches)
- Instructions per cycle
  - (graduated Instructions)/cycles

Derived Metrics for Memory Hierarchy

- **TLB misses per cycle**
  - (data TLB misses + instruction TLB misses)/cycles
- **Avg number of loads per TLB miss**
  - (load count)/(TLB misses)
- **Total L_k data cache accesses**
  - L_{k-1} load misses + L_{k-1} store misses
- **Accesses from L_k per cycle**
  - (L_{k-1} load misses + L_{k-1} store misses)/cycles
- **L_k traffic (analogously memory traffic)**
  - (L_{k-1} load misses + L_{k-1} store misses) * (L_{k-1} cache line size)
- **L_k bandwidth consumed (analogously memory bandwidth)**
  - (L_k traffic)/(wall clock time)

Counting Events with Calipers

- Augment code with
  - —Start counter
  - —Read counter
  - —Stop counter
- Strengths
  - —Measure exactly what you want, anywhere you want
  - —Can be used to guide run-time adaptation
- Weaknesses
  - —Typically requires manual insertion of counters
  - —Monitoring multiple nested scopes can be problematic
  - —Perturbation can be severe with calipers in inner loops
    - – Cost of monitoring
    - – Interference with compiler optimization

Profiling

- Allocate a histogram: entry for each "block" of instructions
- Initialize: bucket[*] = 0, counter = –threshold
- At each event: counter++
- At each interrupt: bucket[PC]++, counter = –threshold

program

```
...
fldl      (%ecx,%eax,8)
fld       %st(0)
fmull     (%edx,%eax,8)
faddl     -16(%ebp)
fstpl     -16(%ebp)
fmul      %st(0), %st
...
```

counter interrupt occurs

PC histogram

```
...
24786
23921
23781 + 1
24226
24134
23985
...
```

increment histogram bucket

Types of Profiling

- **Time-based profiling**
  - —Initialize a periodic timer to interrupt execution every *t* seconds
  - —Every *t* seconds, service interrupt at regular time intervals
- **Event-based profiling**
  - —Initialize an event counter to interrupt execution
  - —Interrupt every time a counter for a particular event type reaches a specified threshold
    - – e.g. sample the PC every 16K floating point operations
- **Instruction-based profiling (Alpha only)**
  - —presented in a few slides

Benefits of Profiling

- **Key benefits**
  - Provides perspective without instrumenting your program
  - Does not depend on preconceived model of where problems lie
    - often problems are in unexpected parts of the code
      - floating point underflows handled by software
      - Fortran 90 sections passed by copy
      - instruction cache misses
      - …
  - Focuses attention on costly parts of code
- **Secondary benefits**
  - Supports multiple languages, programming models
  - Requires little overhead with appropriate sampling frequency

Event Counter Limitation: Attribution

Attribution of events is especially problematic on out-of-order processors

x87 floating point instructions on a Pentium 4

```
                 #define N (1 << 23)
                 #include <string.h>
                 double a[N],b[N];
                 int main() {
                   double s=0,s2=0; int i;
                   memset(a,0,sizeof(a));
                   memset(b,0,sizeof(b));
skid →    3.8%     for (i = 0; i < N; i++) {
         60.0%         s += a[i] * b[i];
         36.2%         s2 += a[i] * a[i] + b[i] * b[i];
                   }
                   printf("s %d s2 %d\n",s,s2);
                 }
```

More Event Counter Limitations

- Event counter interrupts may be deferred: blind spots
  - —e.g. Alpha PALcode is uninterruptible; attributed after PALcode
- Too few counters
  - —can’t concurrently monitor all events
- Lack of detail
  - —e.g., cache miss lacks service time latency

ProfileMe: Instruction-level Profiling

- Goal: Collect two types of information
  - —Summary statistics over workload: program, procedure, loop
  - —Instruction-level information: average behavior for each
- Approach
  - —Randomly choose instructions to be profiled
  - —Record information during their execution
  - —Aggregate sample results at the instruction level
    - – enables estimation of many interesting metrics
  - —Instruction-level information can be aggregated to loops, etc
- Advantages
  - —Low overhead
  - —Completely eliminates difficulties with attribution
    - – even for out-of-order processors
  - —No blind spots
  - —Supports concurrent monitoring
  - —Provides latency detail in addition to events.

ProfileMe Hardware Support

- Select instructions to be profiled
  - —sample fetched instructions rather than only retired ones
    - – use software writable “fetched instruction counter”
    - – decrement counter for each instruction fetched on predicted path
    - – instruction profiled when counter hits zero
  - —enables analysis of when and why instructions abort
- Tag decoded instruction with an index
  - — identify its profile state
- Record information about profiled instructions
  - —see next slide
- Generate an interrupt to deliver information to software

Information About Profiled Instructions

- **Profiled addr space register**
- **Profiled PC register**
- **Profiled address register - effective address of load or store**
- **Profiled event register: bit field**
  - —I-cache miss, d-cache miss, TLB miss, branch taken, branch mispredicted, trap, etc.
- **Profiled path register: code path reaching profiled instruction**
- **Latency registers**
  - —fetch-> map (lack of phys registers, issue queue slots)
  - —map->data ready (data dependences)
  - —data ready -> issue (execution resource contention)
  - —issue -> retire ready (execution latency)
  - —retire ready -> retire (stalls due to prior unretired instructions)
  - —load issue -> completion (memory system latency)

ProfileMe Software Support

- Sample instruction stream randomly
- Service interrupts and log information into profile database
- Analyze data to identify performance problems

Paired Sampling with ProfileMe

- **Problem**
  - —sampling individual instructions is not enough to identify bottlenecks on out-of-order processors
- **Approach**
  - —sample multiple instructions concurrently in flight
    - – instructions in narrow window around target
  - —enables analysis of instruction interactions
    - – obtain statistical estimate of concurrency levels, other measures
- **Question: is paired sampling necessary and useful?**
  - —yes: latency from fetch to retire is not well correlated with wasted issue slots while an instruction is executing
    - – why: varying levels of concurrency