Análise do desempenho de execução de aplicações
(adaptado das aulas 10, 11 e 18 do Bryant)

- "*Análise de desempenho*": para quê?
  - para construir computadores mais rápidos
    - identificação de métricas
      - latência, velocidade, ...
    - ligação entre métricas e factores que influenciam o desempenho
      - $CPU_{Time} = N^{o}_{instr} * CPI * T_{clock}$
  - para melhorar o desempenho das aplicações
    - análise de técnicas de optimização
      - independentes / dependentes da máquina
      - algoritmo / codificação / compilação / *assembly*
    - técnicas de medição
      - escala microscópica / macroscópica
      - uso de *cycle counters* / *interval counting*

Os próximos slides foram retirados da aula do Prof. Bryant

15-213
"The course that gives CMU its Zip!"

Code Optimization I: Machine Independent Optimizations
Sept. 26, 2002

Topics
- Machine-Independent Optimizations
  - Code motion
  - Reduction in strength
  - Common subexpression sharing
- Tuning
  - Identifying performance bottlenecks

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering
  - eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

Machine-Independent Optimizations

  - Optimizations you should do regardless of processor / compiler
- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

Compiler-Generated Code Motion

– Most compilers do a good job with array code + simple loop structures

Code Generated by GCC

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  int *p = a+ni;
  for (j = 0; j < n; j++)
    *p++ = b[j];
}
```

```
  imull %ebx,%eax         # i*n
  movl 8(%ebp),%edi       # a
  leal (%edi,%eax,4),%edx # p = a+i*n (scaled by 4)
# Inner Loop
.L40:
  movl 12(%ebp),%edi      # b
  movl (%edi,%ecx,4),%eax # b+j  (scaled by 4)
  movl %eax,(%edx)        # *p = b[j]
  addl $4,%edx            # p++  (scaled by 4)
  incl %ecx               # j++
  jl .L40                 # loop if j<n
```

Reduction in Strength

– Replace costly operation with simpler one

– Shift, add instead of multiply or divide

```
16*x  -->  x << 4
```

- Utility machine dependent
- Depends on cost of multiply or divide instruction
- On Pentium II or III, integer multiply only requires 4 CPU cycles

– Recognize sequence of products

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    a[n*i + j] = b[j];
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Make Use of Registers

– Reading and writing registers much faster than reading/writing memory

- Limitation
  - – Compiler not always able to determine whether variable can be held in register
  - – Possibility of *Aliasing*
  - – See example later

Machine-Independent Opts. (Cont.)

- Share Common Subexpressions
  - – Reuse portions of expressions
  - – Compilers often not very sophisticated in exploiting arithmetic properties

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n   + j-1];
right = val[i*n   + j+1];
sum = up + down + left + right;
```

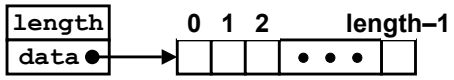
3 multiplications: i*n, (i–1)*n, (i+1)*n

```
int inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

1 multiplication: i*n

```
  leal -1(%edx),%ecx  # i-1
  imull %ebx,%ecx     # (i-1)*n
  leal 1(%edx),%eax   # i+1
  imull %ebx,%eax     # (i+1)*n
  imull %ebx,%edx     # i*n
```

Vector ADT



- Procedures

  `vec_ptr new_vec(int len)`
  - Create vector of specified length

  `int get_vec_element(vec_ptr v, int index, int *dest)`
  - Retrieve vector element, store at *dest
  - Return 0 if out of bounds, 1 if successful

  `int *get_vec_start(vec_ptr v)`
  - Return pointer to start of vector data

  – Similar to array implementations in Pascal, ML, Java
  - E.g., always do bounds checking

Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Procedure
  - Compute sum of all elements of vector
  - Store result at destination location

Time Scales

- Absolute Time
  - Typically use nanoseconds
    - 10^{-9} seconds
  - Time scale of computer instructions
- Clock Cycles
  - Most computers controlled by high frequency clock signal
  - Typical Range
    - 100 MHz
      - 10^8 cycles per second
      - Clock period = 10ns
    - 2 GHz
      - 2 X 10^9 cycles per second
      - Clock period = 0.5ns

Cycles Per Element (1)

– C code for the vector sum functions

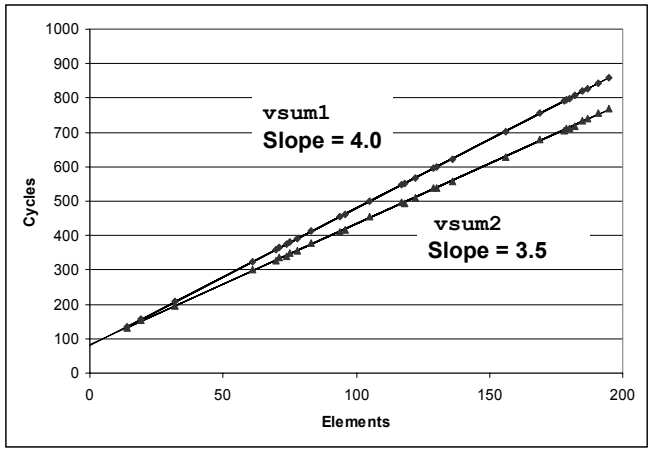
code/opt/vsum.c

```
 1 void vsum1(int n)
 2 {
 3     int i;
 4
 5     for (i = 0; i < n; i++)
 6         c[i] = a[i] + b[i];
 7 }
 8
 9 /* Sum vector of n elements (n must be even) */
10 void vsum2(int n)
11 {
12     int i;
13
14     for (i = 0; i < n; i+=2) {
15         /* Compute two elements per iteration */
16         c[i]   = a[i]   + b[i];
17         c[i+1] = a[i+1] + b[i+1];
18     }
19 }
```

code/opt/vsum.c

Cycles Per Element (2)

- Convenient way to express performance of program that operators on vectors or lists
- Length = n
- T = CPE*n + Overhead



Optimization Example

```
void combine1(vec_ptr v, int *dest)
{
  int i;
  *dest = 0;
  for (i = 0; i < vec_length(v); i++) {
    int val;
    get_vec_element(v, i, &val);
    *dest += val;
  }
}
```

- Procedure
  - Compute sum of all elements of integer vector
  - Store result at destination location
  - Vector data structure and operations defined via abstract data type
- Pentium II/III Performance: Clock Cycles / Element
  - 42.06 (Compiled -g) 31.25 (Compiled -O2)

Os próximos slides foram retirados da aula do Prof. Bryant

15-213

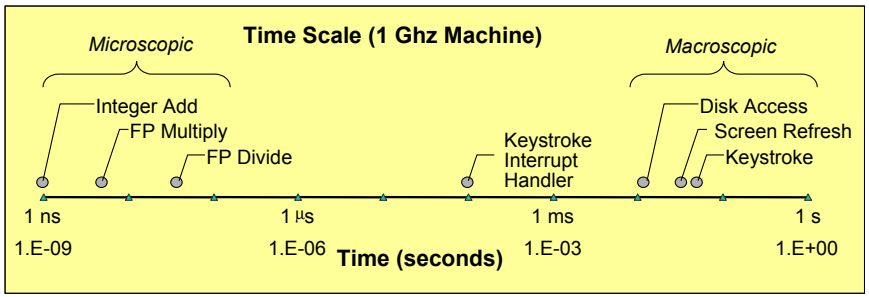
"The course that gives CMU its Zip!"

Time Measurement
Oct. 24, 2002

Topics

- Time scales
- Interval counting
- Cycle counters
- K-best measurement scheme

Computer Time Scales



- Two Fundamental Time Scales
  - Processor: ~10^{-9} sec.
  - External events: ~10^{-2} sec.
    - Keyboard input
    - Disk seek
- Implication
  - Can execute many instructions while waiting for external event to occur
  - Can alternate among processes without anyone noticing

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Maintained as part of process state
    - In Linux, counts elapsed global time
- Special assembly code instruction to access
- On (recent model) Intel machines:
  - 64 bit counter.
  - RDTSC instruction sets `%edx` to high order 32-bits, `%eax` to low order 32-bits

Cycle Counter Period

- Wrap Around Times for 550 MHz machine
  - Low order 32 bits wrap around every 2^{32} / (550 * 10^6) = 7.8 seconds
  - High order 64 bits wrap around every 2^{64} / (550 * 10^6) = 33539534679 seconds
    - 1065 years
- For 2 GHz machine
  - Low order 32-bits every 2.1 seconds
  - High order 64 bits every 293 years

Measuring with Cycle Counter

- Idea
  - Get current value of cycle counter
    - store as pair of unsigned's `cyc_hi` and `cyc_lo`
  - Compute something
  - Get new value of cycle counter
  - Perform double precision subtraction to get elapsed cycles

```
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```

Accessing the Cycle Cntr.

- GCC allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with GCC

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Emit assembly with `rdtsc` and two `movl` instructions

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Instruction String
  - Series of assembly commands
    - Separated by "`;`" or "`\n`"
    - Use "`%%`" where normally would use "`%`"

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Output List
  - Expressions indicating destinations for values `%0`, `%1`, …, `%j`
    - Enclosed in parentheses
    - Must be *lvalue*
      - Value that can appear on LHS of assignment
  - Tag "`=r`" indicates that symbolic value (`%0`, etc.), should be replaced by register

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Input List
  - Series of expressions indicating sources for values `%j+1`, `%j+2`, …
    - Enclosed in parentheses
    - Any expression returning value
  - Tag "`r`" indicates that symbolic value (`%0`, etc.) will come from register

Closer Look at Extended ASM

```
asm("Instruction String"
      : Output List
      : Input List
      : Clobbers List);
}
```

```
void access_counter
  (unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Clobbers List
  - List of register names that get altered by assembly instruction
  - Compiler will make sure doesn't store something in one of these registers that must be preserved across `asm`
    - Value set before & used after

Accessing the Cycle Cntr. (cont.)

- Emitted Assembly Code

```
     movl 8(%ebp),%esi     # hi
     movl 12(%ebp),%edi    # lo
#APP
     rdtsc; movl %edx,%ecx; movl %eax,%ebx
#NO_APP
     movl %ecx,(%esi)      # Store high bits at *hi
     movl %ebx,(%edi)      # Store low bits at *lo
```

  - Used `%ecx` for `*hi` (replacing `%0`)
  - Used `%ebx` for `*lo` (replacing `%1`)
  - Does not use `%eax` or `%edx` for value that must be carried across inserted assembly code

Completing Measurement

  - Get new value of cycle counter
  - Perform double precision subtraction to get elapsed cycles
  - Express as `double` to avoid overflow problems

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  return (double) hi * (1 << 30) * 4 + lo;
}
```

Timing With Cycle Counter

- Determine Clock Rate of Processor
  - Count number of cycles required for some fixed number of seconds

```
double MHZ;
int sleep_time = 10;
start_counter();
sleep(sleep_time);
MHZ = get_counter()/(sleep_time * 1e6);
```

- Time Function P
  - First attempt: Simply count cycles for one

```
double tsecs;
start_counter();
P();
tsecs = get_counter() / (MHZ * 1e6);
```