CS 2461: Computer Architecture 1

Performance Optimization

Next..

- CODE OPTIMIZATION

Course Trivia

- Project 4 due dates are corrected on Blackboard and Website – Nov.24
- Project 3 will now be due December 2nd
  - Changed to Tuesday after thanksgiving
  - Approved teams posted on website (link from main page)
  - If you did not inform me by the Nov.6th deadline, you have to find a team (at most 3) asap…or work on your own.
- "Optional Final Exam"
  - You have completed 100 points of exams
  - If you need to improve your exam score, then the third exam is available to you
  - Exam will be comprehensive
- If you take the final exam then total exam score (used to calculate your grade) will be average % score on the 3 exams: Exam1, Exam 2, Final
- Course grades will be posted by this weekend
  - Check weighted total and scaled total – scaled total is used to assign your final grade
  - Check statistics in blackboard to get an idea of where you stand – or you can meet me for more details.

Code optimization for performance

- A quick look at some techniques that can improve the performance of your code
- Rewrite code to minimize processor cycles
  - But do not mess up the correctness!
  - Reduce number of instructions executed
  - Reduce the "complexity" of instructions
    - In real processors, different arithmetic operations can take different times
- Locality
  - Will improve memory performance

Summary: Memory Access time optimization

- If each access to memory leads to a cache hit then time to fetch from memory is one cycle
  - Program performance is good!
- If each access to memory leads to a cache miss then time to fetch from memory is much larger than 1 cycle
  - Program performance is bad!
- Design Goal:

How to arrange data/instructions so that we have as few cache misses as possible.

Recall CPU time model

CPU time	=	Seconds / Program	=	Instructions / Program	x	Cycles / Instruction	x	Seconds / Cycle

$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

CPU = IC * CPI * Clk

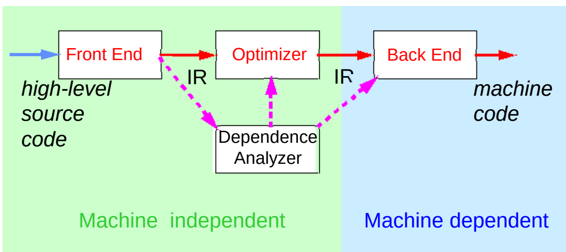
Who can 'change' each parameter

- *CPU = IC * CPI * Clk*
  - *Clk: ?*
  - *IC (number of instructions):?*
  - *CPI: ?*
- *Clk: completely under HW control*
- *IC: programmer and compiler*
- *CPI: compiler and HW*
- *….so what does a compiler do?*

Compiler Tasks

- Code Translation
  - Source language → target language
    - FORTRAN → C
    - C → MIPS, PowerPC or Alpha machine code
    - MIPS binary → Alpha binary
- Code Optimization
  - Code runs faster
  - Match dynamic code behavior to static machine structure

Compiler Structure

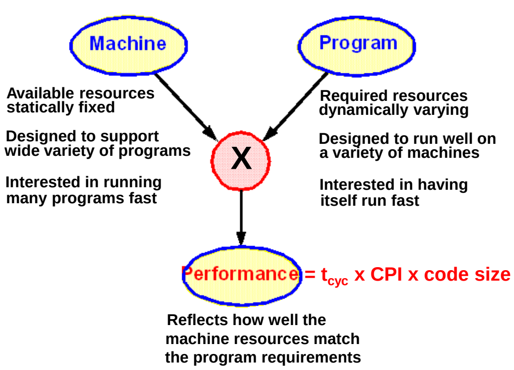


(IR= intermediate representation)

Front End

- Lexical Analysis
  - Misspelling an identifier, keyword, or operator
    e.g. lex
- Syntax Analysis
  - Grammar errors, such as mismatched parentheses
    e.g. yacc
- Semantic Analysis
  - Type checking

Hardware-Software Interface



Formal Model for Code Optimization ?

- Is it a hack job or is there a formal model underlying the various transformations that can help with designing a tool to optimize code ?
  - Need to make sure that transformed code is correct and does not change semantics of the original program.
- Graph theory: model program as a graph (Program dependence graph)
  - Model data and control dependencies
  - Any transformation should give us a homomorphic graph
    - Recall concept of Isomorphism/Homomorphism Discrete Structures courses !!!

The Program Dependence Graph

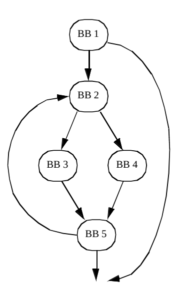
- How to represent control and data flow of a program ?
- The Program Dependence Graph (PDG) is the intermediate (abstract) representation of a program designed for use in optimizations
- It consists of two important graphs:
  - Control Dependence Graph captures control flow and control dependence
  - Data Dependence Graph captures data dependences

Control Flow Graph: Definition

A control flow graph CFG = (N_c ; E_c ; T_c) consists of

- N_c, a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e. a basic block.
- $E_c \subseteq N_c \times N_c \times$ *Labels*, a set of *labeled* edges.

Control Flow Graph




```
main:
      addi r2,  r0,  A     | BB 1
      addi r3,  r0,  B     |
      addi r4,  r0,  C     |
      addi r5,  r0,  N     |
      add  r10, r0,  r0    |
      bge  r10, r5,  end   |
loop:
      lw   r20, 0(r2)      | BB 2
      lw   r21, 0(r3)      |
      bge  r20, r21, T1    |
      sw   r21, 0(r4)      | BB 3
      b    T2              |
T1:
      sw   r20, 0(r4)      | BB 4
T2:
      addi r10, r10, 1     | BB 5
      addi r2,  r2,  4     |
      addi r3,  r3,  4     |
      addi r4,  r4,  4     |
      blt  r10, r5,  loop  |
end:
```


Program behaviour ?

- Model as program dependence graph!
- What is a correct execution ?
  - Execution will only follow valid paths in the program dependence graph!
    - **IF code is written correctly, then force the program to only follow paths in the dependence graph!**
- Note connection to Software security/correctness

Formal Model for Code Optimization ?

- Need to make sure that transformed code is correct and does not change semantics of the original program.
- Graph theory: model program as a graph (Program dependence graph)
  - Model data and control dependencies
- Any transformation should give us a homomorphic graph
  - Recall concept of Isomorphism/Homomorphism Discrete Structures courses !!!

Compiler optimizations

- All 'useful' compilers have code optimizers built into them
  - Optimize time….
  - What about other metrics: power ?
- Machine dependent optimizations
  - Need to know something about the processor details before we can optimize
- Machine independent optimizations
  - These performance improvements to the code are independent of processor specifics

Machine Dependent Optimizations

These need some knowledge of the processor

- Register Allocation
- Instruction Scheduling
- Peephole Optimizations

Peephole Optimizations

- Replacements of assembly instruction through template matching
- Eg. Replacing one addressing mode with another in a CISC

Instruction Scheduling

- Given a source program P, schedule the instructions so as to minimize the overall execution time on the functional units in the target machine
  - This is where ILP processors introduce complexity into the scheduling process
  - Schedule parallel instructions

Register Allocation

- Storing and accessing variables from registers is much faster than accessing data from memory.
  - Variables ought to be stored in registers
- It is useful to store variables as long as possible, once they are loaded into registers
- Registers are bounded in number
  - "register-sharing" is needed over time.
  - Some variables have to be 'flushed' to memory
  - Reading from memory takes longer

Register Allocation

```
{ …
    i=10;
    x= y*z +i;
    while (i<100){
        a = a*100
        b = b +100
        i++;
    }
```

- Suppose you have 3 registers available…
- Can you place a and b into same register ?
- Can you place x and a into same register ?

Register Allocation – Key Concepts

- Determine the range of code over which a variable is used, and determine conflicts between variables
  - Live ranges
  - Using dataflow analysis we can compute live ranges for each variable
- Formulate the problem of assigning variables to registers as a graph problem
  - Graph coloring
  - Use application domain (Instruction execution) to define the priority function

Machine Independent Optimizations

- We will spend some time on these….
- As SW developers, these should be a ‘default’ when you write code…
  - THIS is what separates you from those who take a single programming course and claim they know CS!!

Machine-Independent Optimizations

- Dataflow Analysis and Optimizations
  - Constant propagation
  - Copy propagation
  - Value numbering
- Elimination of common subexpression
- Dead code elimination
- Code motion
- Stength reduction
- Function/Procedure inlining

Code-Optimizing Transformations

- Constant folding

  (1 + 2) $\Rightarrow$ 3

  (100 > 0) $\Rightarrow$ true

Code-Optimizing Transformations

- Copy propagation

  x = b + c
  z = y * x

  $\Rightarrow$

  x = b + c
  z = y * (b + c)

Code-Optimizing Transformations

- Common subexpression

```
x = b * c + 4          t = b * c
z = b * c - 1    ⇒     x = t + 4
                       z = t - 1
```


Code-Optimizing Transformations

- Dead code elimination

```
x = 1
x = b + c
```

or if x is not referred to at all

Code-Optimizing Transformations

- Constant folding

```
(1 + 2)      ⇒   3
(100 > 0)    ⇒   true
```

- Copy propagation

```
x = b + c          x = b + c
z = y * x    ⇒     z = y * (b + c)
```

- Common subexpression

```
x = b * c + 4          t = b * c
z = b * c - 1    ⇒     x = t + 4
                       z = t - 1
```

- Dead code elimination

```
x = 1
x = b + c
```

or if x is not referred to at all

Code Optimization Example

```
x = 1
y = a * b + 3
z = a * b + x + z + 2
x = 3
```

propagation

```
x = 1
y = a * b + 3
z = a * b + 1 + z + 2
x = 3
```

constant folding

```
x = 1
y = a * b + 3
z = a * b + 3 + z
x = 3
```

dead code elimination

```
y = a * b + 3
z = a * b + 3 + z
x = 3
```

common subexpression

```
t = a * b + 3
y = t
z = t + z
x = 3
```

Code Motion

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop
- Move code between blocks
  - eg. move loop invariant computations outside of loops
- What does this reduce ?

```
while ( i < 100 ) {
        *p = x / y + i
        i = i + 1
}
```

```
t = x / y
while ( i < 100 ) {
        *p = t + i
        i = i + 1
}
```

CS 2461

Code motion example

```
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];
}
```

CS 2461

- 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];
}
```

CS 2461

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  `16*x   -->   x << 4`
  - Utility is 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;
}
```

CS 2461

Strength Reduction

- Replace complex (and costly) expressions with simpler ones
  - What does this reduce ?
- E.g.

```
a : = b*17          a: = (b<<4) + b
```

- E.g.

```
                         p  =  & a[ i ]
                         t  =  i * 100
while ( i < 100 ) {      while ( i < 100 ) {
      a[ i ] =  i * 100        *p  =  t
      i  =  i + 1              t  =  t + 100
}                              p  =  p  +  4
                               i  =  i  + 1
                         }
```

loop invariant: &a[i]==p, i*100==t

Function Inlining

- What happens on a function call ?
  - How are function calls implemented on the machine ?
  - Is function call = one subroutine call ?
- Function call in C = number of instructions in machine code
  - Create activation records, allocate memory
  - Manipulate stack and frame pointers
- What happens if we replace function call with body of function ?
  - Inline the function

Function Inlining

```
…                      int myfunc(int m,n)
x= myfunc(i,j)         {
…                            return(m+n);}
```

After inlining:

```
…
x = m+n
…..
```


Link with Memory organization… . .

- Let’s use array data structures to guide our discussions
- Recall: accesses to cache better than accesses to main memory/disk
- Recall: Multidimensional Arrays

Declaration

```
int ia[3][4];
```

Type

Address

Number of Rows

Number of Columns

Declaration at compile time i.e. size must be known

How does a two dimensional array work?

	0	1	2	3
0				
1				
2				

How would you store it?

How would you store it?

	0	1	2	3
0				
1				
2				

Column Major Order

0,0	1,0	2,0	0,1	1,1	2,1	0,2	1,2	2,2	0,3	1,3	2,3

Column 0 Column 1 Column 2 Column 3

Row Major Order

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

Row 0 Row 1 Row 2

Advantage of row major

- Using Row Major Order allows visualization as an array of arrays
- C stores arrays in Row Major order

ia[1]

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

ia[1][2]

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

Locality of Access

- How are elements in the array accessed in your program ?
  - Row major
  - How would you iterate over the 2-D array to maintain locality ?

CS 2461

Locality

- Principle of Locality:
  - Programs tend to reuse data and instructions near those they have used recently, or that were recently referenced themselves.
  - Temporal locality: Recently referenced items are likely to be referenced in the near future.
  - Spatial locality: Items with nearby addresses tend to be referenced close together in time.

CS 2461

Locality and performance

- Recall: Memory = Cache + Main memory
  - Cache contains small number of bytes
- Recall: cache is arranged as a set of blocks
  - Can only fetch block at a time
- Example:
  - Assume each cache block has 4 words
  - If you fetch a block with addresses {0,1,2,3}
  - If four successive instructions use locations 0,1,2,3 then we only have one cache miss (first time to fetch block into cache)
  - If four successive instructions use locations 0,4,8,12 then each time we have to fetch a new cache block
    - Each memory access is a access to main memory
- Goal: have locality in memory accesses in the cache

CS 2461

Locality

- Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer.

CS 2461

Locality Example

- Question: Does this function have good locality?

```
int sumarraycols(int a[M][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum
}
```


Locality Example

- Question: Does this function have good locality?

```
int sumarrayrows(int a[M][N])
{
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum
}
```


Improving Memory Access Times (Cache Performance) by Compiler Optimizations

- McFarling [1989] improve perf. By rewriting the software
- Instructions
  - Reorder procedures in memory so as to reduce cache misses
  - Code Profiling to look at cache misses(using tools they developed)
- Data
  - *Merging Arrays*: improve spatial locality by single array of compound elements vs. 2 arrays
  - *Loop Interchange*: change nesting of loops to access data in order stored in memory
  - *Loop Fusion*: Combine 2 independent loops that have same looping and some variables overlap
  - *Blocking*: Improve temporal locality by accessing "blocks" of data repeatedly vs. going down whole columns or rows

Compiler optimizations – merging arrays

- This works by improving spatial locality
- For example, some programs may reference multiple arrays of the same size at the same time
  - Could be bad – not enough locality
    - Accesses may interfere with one another in the cache – conflict misses
- A solution: Generate a single, compound array…

```
/* Before:*/
int tag[SIZE]
int byte1[SIZE]
int byte2[SIZE]
int dirty[size]
```

```
/* After */
struct merge {
        int tag;
        int byte1;
        int byte2;
        int dirty;
}
struct merge cache_block_entry[SIZE]
```

Merging Arrays Example

```
/* Before: 2 sequential arrays */
int val[SIZE];
int key[SIZE];

/* After: 1 array of stuctures */
struct merge {
  int val;
  int key;
};
struct merge merged_array[SIZE];
```

Reducing conflicts between val & key; improve spatial locality

Compiler optimizations – loop interchange

- Some programs have nested loops that access memory in non-sequential order
  - Simply changing the order of the loops may make them access the data *in* sequential order…
- What's an example of this?
  - Recall: C stores 2-D arrays in row-major format

Loop Interchange Example

```
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
      for (i = 0; i < 5000; i = i+1)
            x[i][j] = 2 * x[i][j];
```


Loop Interchange Example

```
/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
      for (j = 0; j < 100; j = j+1)
            x[i][j] = 2 * x[i][j];
```

Sequential accesses instead of striding through memory every 100 words; improved spatial locality

Compiler optimizations – loop fusion

- This one's pretty obvious once you hear what it is…
- Seeks to take advantage of:
  - Programs that have separate sections of code that access the same arrays in different loops
    - Especially when the loops use common data
  - The idea is to "fuse" the loops into one common loop
- What's the target of this optimization?

Loop Fusion Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      a[i][j] = 1/b[i][j] * c[i][j];
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
      d[i][j] = a[i][j] + c[i][j];
```

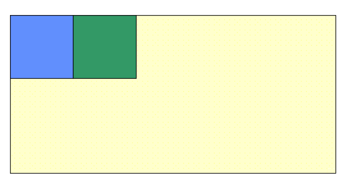

Loop Fusion Example

```
/* After */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
  {   a[i][j] = 1/b[i][j] * c[i][j];
      d[i][j] = a[i][j] + c[i][j];}
```

2 misses per access to a & c vs. one miss per access; improve spatial locality

Compiler Optimization: Blocking.

- Can you keep locality in all memory operations
- This is probably the most "famous" of compiler optimizations to improve cache performance
- Another common concept: blocking
  - Rewrite code to process blocks of data at a time
  - Size of block = ???? Size of cache block!!



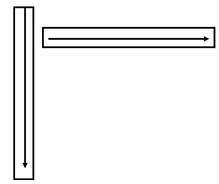
Compiler optimizations – blocking

- Tries to reduce misses by improving temporal locality and spatial locality
- To get a handle on this, you have to work through code on your own
- this is used mainly with arrays!
- Simplest case??
  - Row-major access

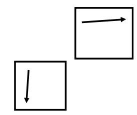
CS 2461

Blocking Example

```
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
     {r = 0;
      for (k = 0; k < N; k = k+1){
        r = r + y[i][k]*z[k][j];};
      x[i][j] = r;
     };
```



- Two Inner Loops:
  - Read all NxN elements of z[]
  - Read N elements of 1 row of y[] repeatedly
  - Write N elements of 1 row of x[]



- Idea: compute on BxB submatrix that fits

CS 2461

Code Optimization and Compilers

- Modern compilers provide a menu of code optimization features
  - Inlining, strength reduction, register allocation, loop optimizations, etc.
- Some provide default optimization levels
  - Example: gcc -03 test.c
- Bottom Line: Everyone wants to run optimized code
  - Being smart with your solution!

CS 2461

Example of Code Optimization: Project 5

- Topic: Performance Optimization
  - Given code for Image Smoothing and Image Rotation, rewrite the code to make it run faster.
    - Use only techniques covered in class.
- Will be handed out after thanksgiving break – due December 7th 10am
  - Should take you 4-6 hours to complete

CS 2461