Performance I

CS 105: Computer Systems
Lecture 24

Melissa O'Neill

April 22, 2026

Learning Goals

- **Describe ways to optimize code to improve performance**
  - *There's more to performance than asymptotic complexity*
  - Examples of **machine independent** optimizations
- **Understand examples of optimization blockers for compilers**
  - Procedure calls
  - Memory aliasing
- **Begin building understanding for machine *de*pendent optimizations**

Example: memory aliasing

- **Consider procedure `setVals`. Couldn't compiler just change assignment of `t` to *save a reference to memory?***

```
void setVals(int* q, int* p){
  int x = 1000;
  int y = 300;
  int t = 0;

  *q = y;
  *p = x;
  t = *q; /* why can't compiler
             change this to t = y? */
  printf("t = %d \n",t);
}
```

Exercise:

Ideas to improve the performance of these three functions?

(1)
```
/* Copy nxn matrix a into b */
void matrixCopy(int* a, int* b, int n)
  for (j = 0; j < n; j++)
    for (i = 0; i < n; i++) {
        b[i][j] = a[i][j];
}
```

(2)
```
/* Change each character in string
    to lower case */
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

(3)
```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
          b[i] = 0;
          for (j = 0; j < n; j++)
              b[i] += a[i][j];
    }
}
```


Code Motion

Move unnecessary code out of loops

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

`n*i` doesn't need to be computed in the inner loop!

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


Reduction in Strength

Reduce slow operations by faster ones
(e.g., addition is faster than multiplication which is faster than division)

n*i multiplication can be achieved with addition

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

Optimizing Compilers

- **Provide efficient mapping of program to machine code**
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- **-O flag sets optimization level**
  (we experimented with these in the Debugger lab!)

Rear Admiral **Grace Hopper invented first compiler!**

Fran Allen developed many optimizing compilation techniques

Constant folding

- **Compiler can do arithmetic and "fold" constants**
- **Example:**

```
int main(int argc, char *argv[]) {
  int sum = 2 + 4 + 6 + 8;
  int len = strlen("Hello world");
  printf("%d %d\n", len, sum);
  return 0;
}
```

```
assembler code for function main:
   sub    $0x8,%rsp
   mov    $0x14,%ecx
   mov    $0xb,%edx
   xor    %eax,%eax
   lea    0xfad(%rip),%rsi    # format string: %d %d\n
   mov    $0x1,%edi
   call   0x401030 <__printf_chk@plt>
   xor    %eax,%eax
   add    $0x8,%rsp
   ret
```


Function inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for other optimizations
  - Can make code bigger and therefore slower (size; i-cache)
- **Example from the Debugger lab! (no call to `loop_while` in `main`)**

```
int loop_while(int a, int b)
{
  int i = 0;
  int result = a;
  while (i < 256) {
    result += a;
    a -= b;
    i += b;
  }
  return result;
}
int main(int argc, char *argv[]) {
  printf("%d\n",
         loop_while(atoi(argv[1]), 16));
  return 0;
}
```

```
assembler code for function main:
…                          # rax: a
lea     -0x10(%rax),%ecx   # ecx: a - 16
mov     %ecx,%edx          # edx: a - 16
shl     $0x4,%edx          # edx: 16(a-16)
sub     %ecx,%edx          # edx: 15(a-16)
lea     -0x690(%rdx,%rax,2),%edx
        # → edx: -1680 + 15(a-16) + 2a
…
```


Dead code elimination

- **Don't emit code that will never be executed or whose result is overwritten**
- **Examples:**

```
if (0)
  printf("Hello!");
else
  printf("Goodbye");
```

```
x = 18;
x= 42;
```

- **These may look silly, but…**
  - Can be produced by other optimizations
  - Assignments to x might be far apart

Functions may have side effects

```
/* My version of strlen */
size_t strlen(const char *s)
{
  static num=0;
  num++;
  size_t length = 0;
  while (*s != '\0') {
    s++;
    length++;
  }
  return length;
}
```

Compilers won't remove function call in order to optimize

Reduce memory accesses

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
       b[i] = 0;
       for (j = 0; j < n; j++)
           b[i] += a[i*n + j];
    }
}
```

Memory access is slow

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
       double bi = 0;
       for (j = 0; j < n; j++)
           bi += a[i*n + j];
        b[i] = bi;
    }
}
```

Allows compiler to save `bi` in register.

Why doesn't compiler do this optimization automatically?

Reduce memory accesses – Example

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n){
    long i, j;
    for (i = 0; i < n; i++) {
       b[i] = 0;
       for (j = 0; j < n; j++)
           b[i] += a[i*n + j];
    }
}
```

```
double a[9] =
  { 0,   1,   2,
    4,   8,  16},
   32,  64, 128};
double b[3] = [4, 8, 16];
sum_rows1(a, b, 3);
```

Value of `b`:

```
init:  b=[4, 8, 16]
```

```
When loop ends for
i = 0: b=[3, 8, 16]
```

```
When loop ends for
i = 1: b=[3, 28, 16]
```

```
When loop ends for
i = 2: b=[3, 28, 224]
```

- **Code updates `b[i]` on every iteration**
  - Compiler must consider possibility that these updates will affect program behavior

Exercise: fill in values of b for i=0, 1, 2

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```

```
double a[9] =
  { 0,  1,  2,
    4,  8,  16},
   32,  64, 128};
double *b = a+3;
sum_rows1(a, b, 3);
```

Value of b:

```
init:  b=[4, 8, 16]
```

```
When loop ends for
i = 0: b=
```

```
When loop ends for
i = 1: b=
```

```
When loop ends for
i = 2: b=
```


The `restrict` Keyword

- **Tells the compiler "this pointer never aliases"**
  - The compiler trusts you!
  - Not "officially" part of C++, only C, but all C++ compilers support it as `__restrict`.

```
/* Sum rows of n X n matrix a
   and store in vector b  */
void sum_rows1(double *restrict a, double *restrict b, long n)
{
    long i, j;
    for (i = 0; i < n; i++) {
        b[i] = 0;
        for (j = 0; j < n; j++)
            b[i] += a[i*n + j];
    }
}
```


Exercise

What are some optimizations we can make to procedure `combine`?

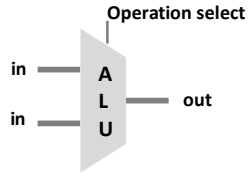
```
/* determine average value of an int array */
int findAvg(int *a, int length);

/* sum some elements of an array based on its avg value */
void combine(int* a, int length, int *dest) {
  unsigned int i;
  *dest = 0;

  for (i = 0; i < length; i++) {
    printf(a[i]);
    if (a[i] > findAvg(a, length));
      *dest = *dest + a[i];
  }
}
```

Machine Dependent Optimizations

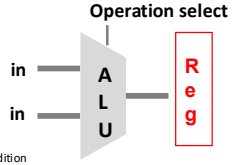
- **Techniques before this: machine *independent***
- **Machine dependent techniques**
  - Use knowledge of underlying system design
  - Examples: instruction pipelining, customizing code for a particular cache configuration

Combinational Logic and Clocked Registers

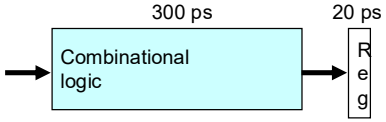
- **Recall: ALU is a combinational logic circuit**
  - Boolean logic, output is a function of the input(s)

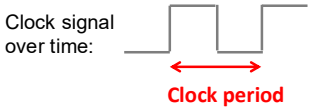


- **Executing machine instructions**
  - First decode an instruction:
    - Two operands as **inputs**, e.g., two register values
    - The operation to perform, e.g., `add`
  - After some delay, result appears as **output**
  - Inputs change for next instruction… and repeat…
- **Save output in *clocked register* before it changes**



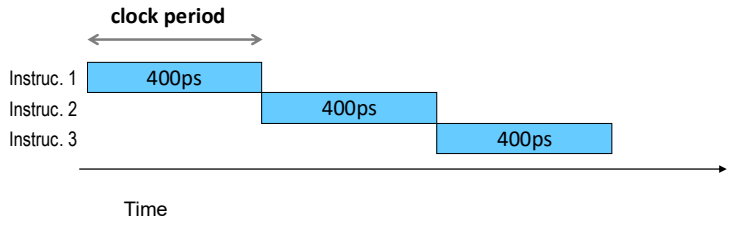
Combinational Logic Latency and Throughput





- **Timing**
  - System clock triggers storing ("latching") of results into register
  - Goal: store result of the combinational logic when it's done
  - **Latency** (**delay**) is the time from input to finish storing output
- **Example**
  - Above, clock period has to be at least 320 picoseconds (ps)
  - The clock speed is at best $1/(320 \times 10^{-12})$ cycles per second ≈3.12 GHz
  - Best **throughput** would be $1/(320 \times 10^{-12})$ instructions per second ≈3.12 GIPS

Timing Diagrams



- **Cannot start new instruction until previous one completes**
  - If each rectangle above has a delay of 400ps (including both logic and register storing), what's the throughput?

1/400ps

Exercise

In the combinational logic below, Circuit 3 takes as input the outputs from Circuits 1 and 2.

What is the **shortest clock period** the system can use if the CPU contains the following circuit? What is the **best throughput** it can achieve?