COMP2310/COMP6310
Systems, Networks, & Concurrency

Convener: Prof John Taylor

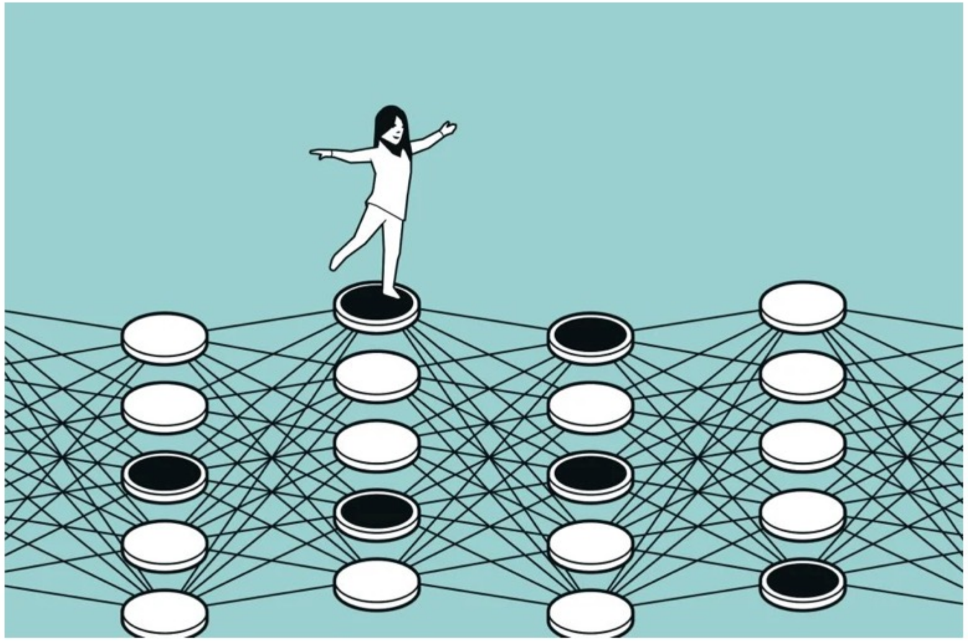
Course Update

- **Assignment 1 – Marking now**
- **Checkpoint 2 (7.5%) – This week**
  - Attend the lab as per Checkpoint 1
- **Quiz 2 (5%) – Next week**
  - Cover all material in lectures and labs weeks 6-10
  - Bring one double-sided A4 sheet of notes
- **Final Exam – Closed Book**
  - Wednesday 12/11/2025 2-5:15pm
  - Melville Hall

The Nobel Prize in Physics 2024

They used physics to find patterns in information

This year's laureates used tools from physics to construct methods that helped lay the foundation for today's powerful machine learning. John Hopfield created a structure that can store and reconstruct information. Geoffrey Hinton invented a method that can independently discover properties in data and which has become important for the large artificial neural networks now in use.

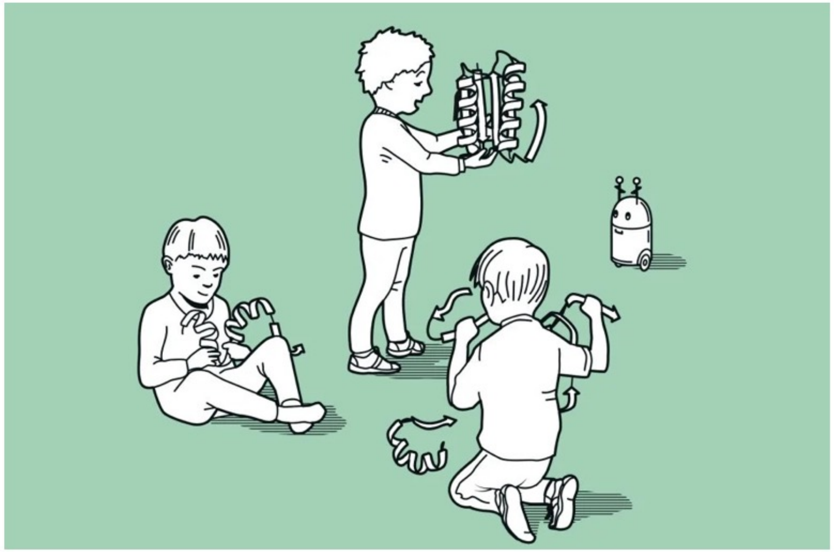


"for foundational discoveries and inventions that enable machine learning with artificial neural networks"

The Nobel Prize in Chemistry 2024

They cracked the code for proteins' amazing structures

The Nobel Prize in Chemistry 2024 is about proteins, life's ingenious chemical tools. David Baker has succeeded with the almost impossible feat of building entirely new kinds of proteins. Demis Hassabis and John Jumper have developed an AI model to solve a 50-year-old problem: predicting proteins' complex structures. These discoveries hold enormous potential.



- One half of the prize goes to Demis Hassabis and John Jumper, who have utilised artificial intelligence to successfully solve a problem that chemists wrestled with for over 50 years: predicting the three-dimensional structure of a protein from a sequence of amino acids. This has allowed them to predict the structure of almost all 200 million known proteins.
- The other half of the prize is awarded to David Baker. He has developed computerised methods for achieving what many people believed was impossible: creating proteins that did not previously exist and which, in many cases, have entirely new functions.

Program Optimization

Acknowledgement of material: With changes suited to ANU needs, the slides are obtained from
Carnegie Mellon University: https://www.cs.cmu.edu/~213/

Today

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Machine-dependent optimization
- Benchmark example

Back in the Good Old Days, when the term "software" sounded funny and Real Computers were made out of drums and vacuum tubes, Real Programmers wrote in machine code.

Not FORTRAN. Not RATFOR. Not, even, assembly language.

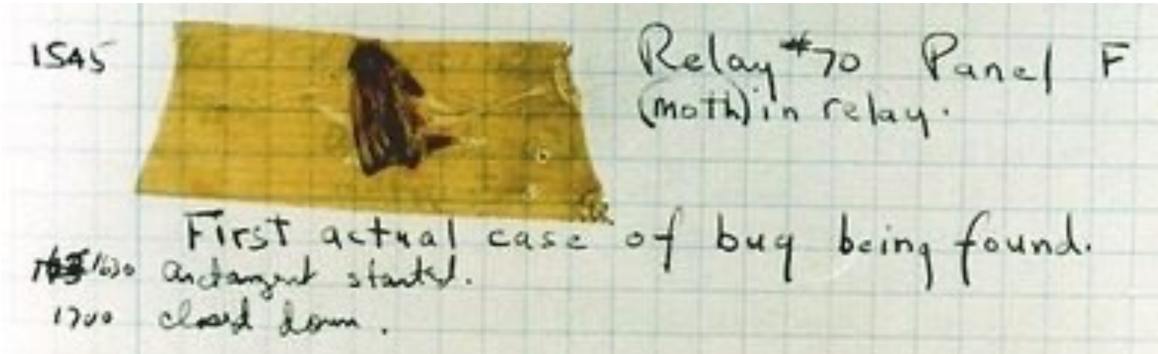
Machine Code.

Raw, unadorned, inscrutable hexadecimal numbers. Directly.

— “The Story of Mel, a Real Programmer”
Ed Nather, 1983

Rear Admiral Grace Hopper

- First person to find an actual bug (a moth)
- Invented first compiler in 1951 (precursor to COBOL)
- "I decided data processors ought to be able to write their programs in English, and the computers would translate them into machine code"



John Backus

- Developed FORTRAN in 1957 for the IBM 704
- Oldest machine-independent programming language still in use today
- “Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, I started work on a programming system to make it easier to write programs”

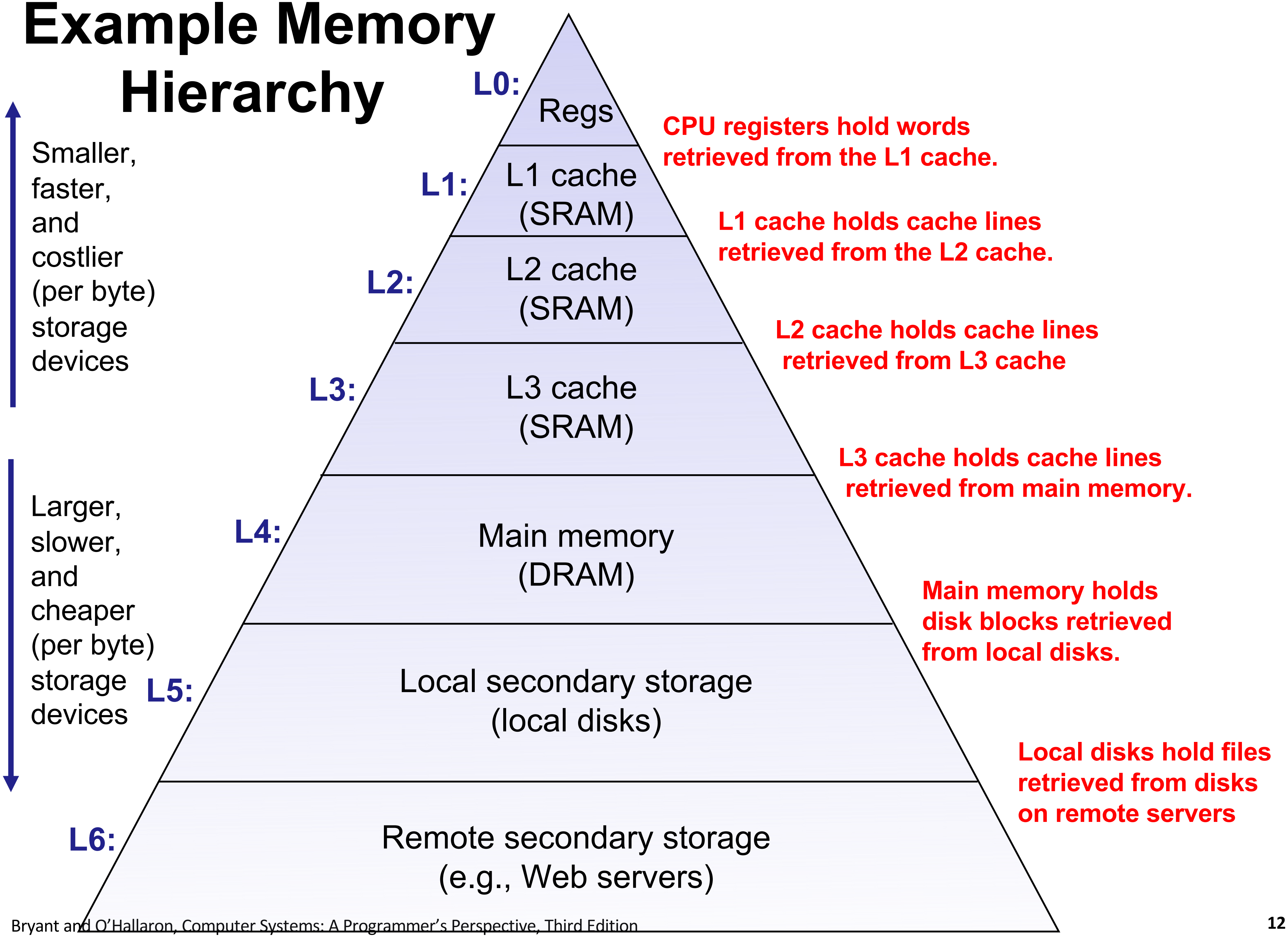
Fran Allen

- Pioneer of many optimizing compilation techniques
- Wrote a paper in 1966 that introduced the concept of the control flow graph, which is still central to compiler theory today
- First woman to win the ACM Turing Award

Goals of compiler optimization

- **Minimize number of instructions**
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once
- **Avoid branching**
  - Don’t make unnecessary decisions at all
  - Make it easier for the CPU to predict branch destinations
  - “Unroll” loops to spread cost of branches over more instructions

Example Memory Hierarchy

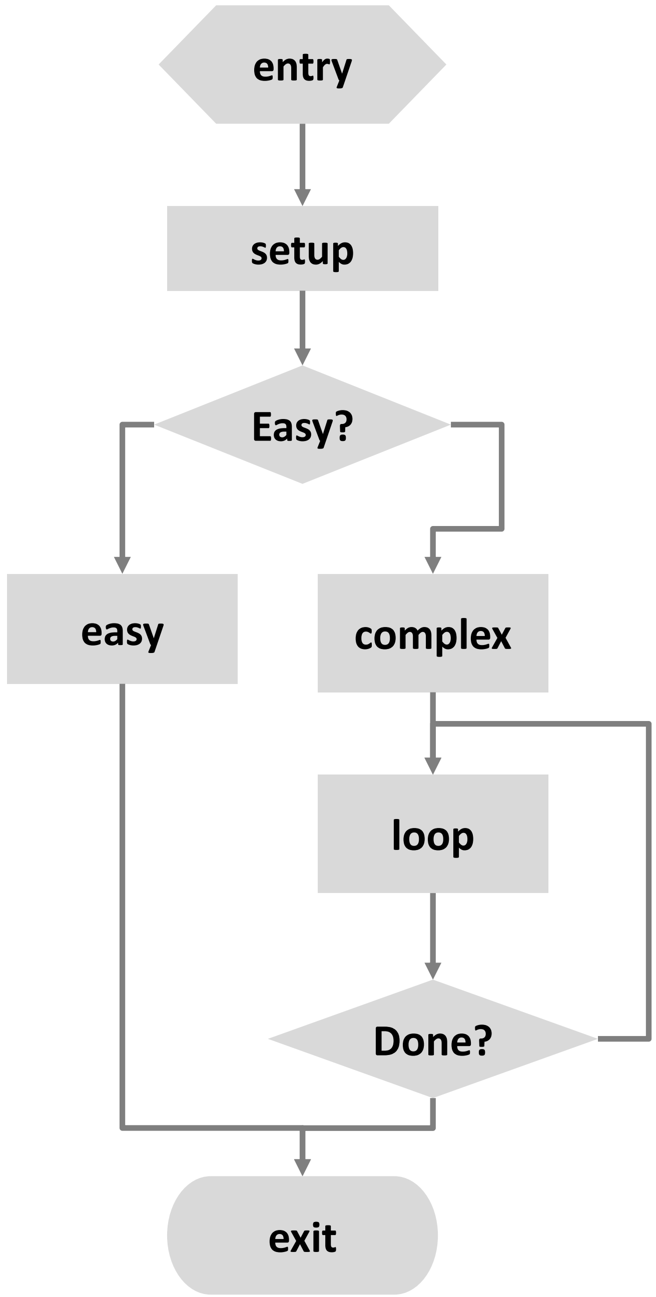


Limits to compiler optimization

- **Generally cannot improve algorithmic complexity**
  - Only constant factors, but those can be worth 10x or more…
- **Must not cause *any* change in program behavior**
  - Programmer may not care about "edge case" behavior, but compiler does not know that
  - Exception: language may declare some changes acceptable
- **Often only analyze one function at a time**
  - Whole-program analysis ("LTO") expensive but gaining popularity
  - Exception: *inlining* merges many functions into one
- **Tricky to anticipate run-time inputs**
  - Profile-guided optimization can help with common case, but…
  - "Worst case" performance can be just as important as "normal"
  - Especially for code exposed to *malicious* input (e.g. network servers)

Two kinds of optimizations

- **Local optimizations work inside a single *basic block***
  - Constant folding, strength reduction, dead code elimination, (local) CSE, …
- **Global optimizations process the entire *control flow graph* of a function**
  - Loop transformations, code motion, (global) CSE, …



Today

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Machine-dependent optimization
- Benchmark example

Next several slides can be done live…

- https://godbolt.org/z/Es5s8qsvj

- Go to Godbolt (the compiler explorer) to play around with C and the resulting assembly generated under different compiler optimizations (change the flag from –O3 to –Og, etc. to see more or less aggressive optimization).

Constant folding

- **Do arithmetic in the compiler**

```
long mask = 0xFF << 8;    →
long mask = 0xFF00;
```

- **Any expression with constant inputs can be folded**
- **Might even be able to remove library calls…**

```
size_t namelen = strlen("Harry Bovik");    →
size_t namelen = 11;
```

Dead code elimination

- **Don’t emit code that will never be executed**

  ~~`if (0) { puts("Kilroy was here"); }`~~
  ~~`if (1) {`~~ `puts("Only bozos on this bus");` ~~`}`~~

- **Don’t emit code whose result is overwritten**

  ~~`x = 23;`~~
  `x = 42;`

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

Common subexpression elimination

- **Factor out repeated calculations, only do them once**

```
norm[i] = v[i].x*v[i].x + v[i].y*v[i].y;
  →
elt = &v[i];
x = elt->x;
y = elt->y;
norm[i] = x*x + y*y;
```

Code motion

- **Move calculations out of a loop**
- **Only valid if every iteration would produce same result**

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

→

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

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower (size; i-cache)

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true **Does nothing** **Can constant fold**

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

Today

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Machine-dependent optimization
- Benchmark example

Memory Aliasing

```
/* 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];
    }
}
```

```
        movq    $0, (%rsi)
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%rdi), %xmm0
        movsd   %xmm0, (%rsi)
        addq    $8, %rdi
        cmpq    %rcx, %rdi
        jne     .L4
```

- Code updates `b[i]` on every iteration
- Why couldn't compiler optimize this away?

Memory Aliasing

```
/* 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] = A+3;

sum_rows1(A, B, 3);
```

```
double A[9] =
  { 0,   1,   2,
    3,  22, 224},
   32,  64, 128};
```

Value of B:

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

i = 0: [3, 8, 16]

i = 1: [3, 22, 16]

i = 2: [3, 22, 224]
```

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

Avoiding Aliasing Penalties

```
/* Sum rows of n X n matrix a and store in vector b. */
void sum_rows2(double *a, double *b, long n) {
    long i, j;
    for (i = 0; i < n; i++) {
        double val = 0;
        for (j = 0; j < n; j++)
            val += a[i*n + j];
        b[i] = val;
    }
}
```

```
        pxor    %xmm0, %xmm0
.L4:
        addsd   (%rdi), %xmm0
        addq    $8, %rdi
        cmpq    %rax, %rdi
        jne     .L4
        movsd   %xmm0, (%rsi)
```

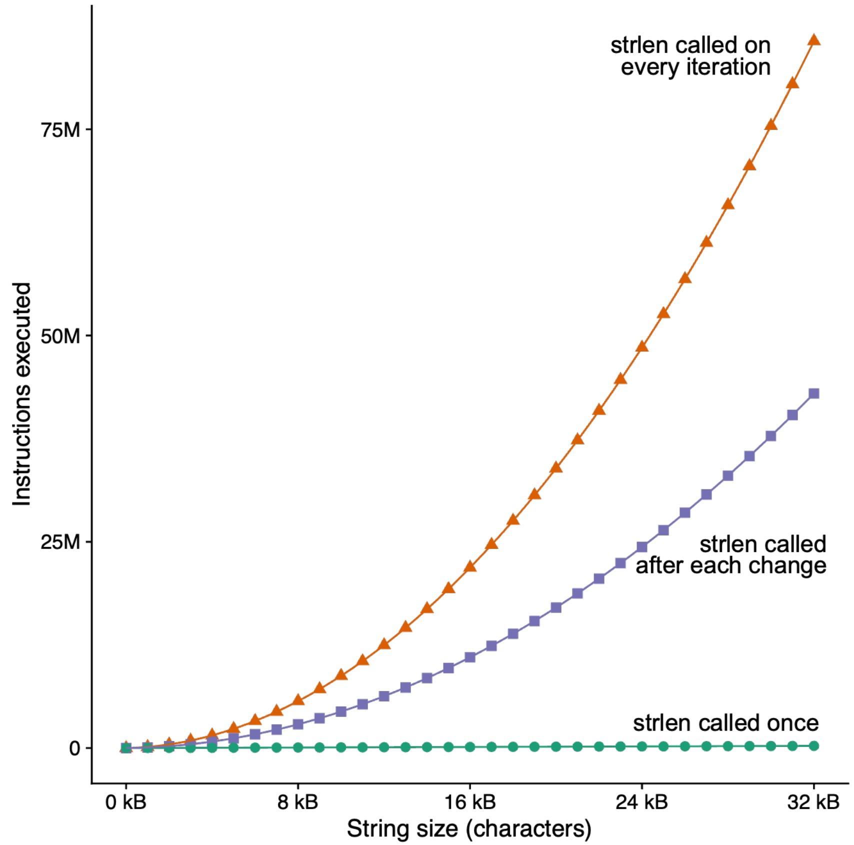
- Use a local variable for intermediate results
- Use restrict keyword
  - Tells compiler that this is the “only” pointer to that memory location

Move function calls out of loops

```
void lower_quadratic(char *s) {
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] += 'a' - 'A';
}

void lower_still_quadratic(char *s) {
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] += 'a' - 'A';
      n = strlen(s);
    }
}

void lower_linear(char *s) {
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] += 'a' - 'A';
}
```



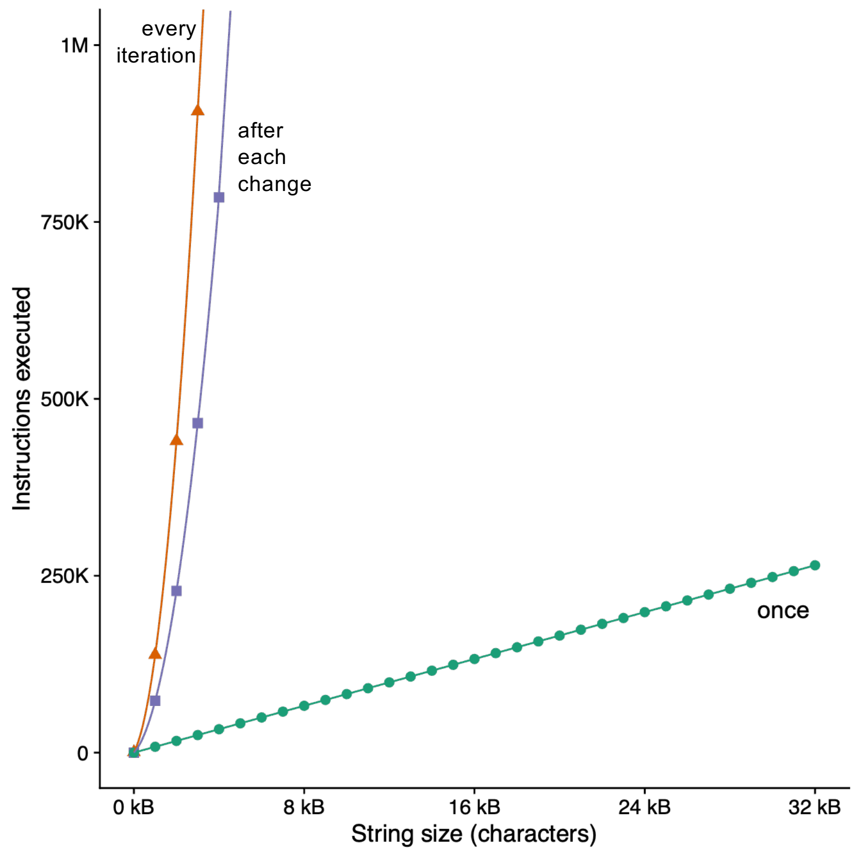
Lots more examples of this kind of bug: accidentallyquadratic.tumblr.com

Can’t move function calls out of loops

```
void lower_quadratic(char *s) {
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] += 'a' - 'A';
}

void lower_still_quadratic(char *s) {
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z') {
      s[i] += 'a' - 'A’;
      n = strlen(s);
    }
}

void lower_linear(char *s) {
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] += 'a' - 'A';
}
```



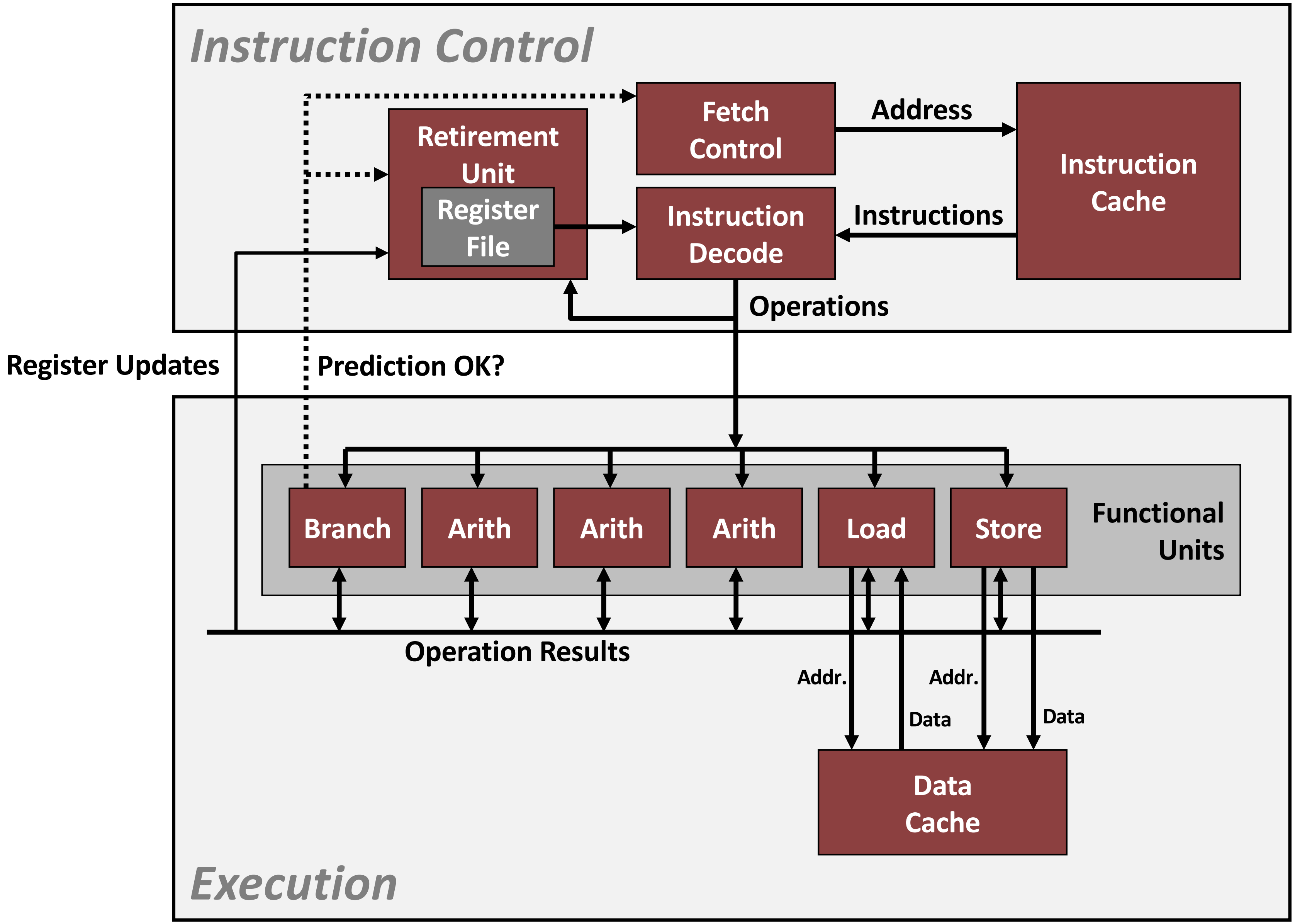
Strength Reduction

- x = y * 4 → x = y << 2
- Replace expensive operations with cheaper ones

Today

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Machine-dependent optimization
- Benchmark example

Modern CPU Design



Branches Are A Challenge

- **Instruction Control Unit** must work well ahead of **Execution Unit** to generate enough operations to keep EU busy

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Executing

Need to know which way to branch …

If the CPU has to wait for the result of the cmp before continuing to fetch instructions, may waste tens of cycles doing nothing!

Branch Prediction

- ***Guess* which way branch will go**
  - Begin executing instructions at predicted position
  - But don't actually modify register or memory data

```
404663:   mov     $0x0,%eax
404668:   cmp     (%rdi),%rsi
40466b:   jge     404685
40466d:   mov     0x8(%rdi),%rax

  . . .

404685:   repz retq
```

Predict Taken

Continue Fetching Here

Branch Prediction Through Loop

```
401029:  mulsd   (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```
i = 98

```
401029:  mulsd   (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```
i = 99

```
401029:  mulsd   (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```
i = 100

```
401029:  mulsd   (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```
i = 101

Assume
array length = 100

Predict Taken (OK)

Predict Taken (Oops)

Read invalid location

Executed

Fetched

Branch Misprediction Invalidation

Assume
array length = 100

```
401029:  mulsd   (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 98

Predict Taken (OK)

```
401029:  mulsd   (%rdx),%xmm0,%xmm0
40102d:  add     $0x8,%rdx
401031:  cmp     %rax,%rdx
401034:  jne     401029
```

i = 99

Predict Taken (Oops)

~~401029: mulsd   (%rdx),%xmm0,%xmm0~~
~~40102d: add $0x8,%rdx~~
~~401031: cmp   %rax,%rdx~~
~~401034: jne   401029~~

i = 100

~~401029: mulsd   (%rdx),%xmm0,%xmm0~~
~~40102d: add $0x8,%rdx~~
~~401031: cmp   %rax,%rdx~~
~~401034: jne   401029~~

i = 101

Invalidate

Branch Misprediction Recovery

```
401029:   mulsd   (%rdx),%xmm0,%xmm0
40102d:   add     $0x8,%rdx
401031:   cmp     %rax,%rdx
401034:   jne     401029
401036:   jmp     401040
  . . .
401040:   movsd   %xmm0,(%r12)
```

i = 99

Definitely not taken

Reload Pipeline

- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

Branch Prediction Numbers

- **A simple heuristic:**
  - Backwards branches are often loops, so predict taken
  - Forwards branches are often ifs, so predict not taken
  - >95% prediction accuracy just with this!
- **Fancier algorithms track behavior of each branch**
  - Subject of ongoing research
  - 2011 record (https://www.jilp.org/jwac-2/program/JWAC-2-program.htm): 34.1 mispredictions per 1000 instructions
  - Current research focuses on the remaining handful of "impossible to predict" branches (strongly data-dependent, no correlation with history)
    - e.g. https://hps.ece.utexas.edu/pub/PruettPatt_BranchRunahead.pdf
- **Deep Learning https://arxiv.org/abs/2112.14911**

Optimizing for Branch Prediction

- **Reduce # of branches**
  - Transform loops
  - Unroll loops
  - Use conditional moves
    - Not always a good idea
- **Make branches predictable**
  - Sort data
    https://stackoverflow.com/questions/11227809
  - Avoid indirect branches
    - function pointers
    - virtual methods

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    leal   -65(%rdx), %ecx
    cmpb   $25, %cl
    ja     .Lskip
    addl   $32, %edx
    movb   %dl, 0(%rbp,%rbx)
.Lskip:
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

```
.Loop:
    movzbl 0(%rbp,%rbx), %edx
    movl   %edx, %esi
    leal   -65(%rdx), %ecx
    addl   $32, %edx
    cmpb   $25, %cl
    cmova  %esi, %edx
    movb   %dl, 0(%rbp,%rbx)
    addl   $1, %rbx
    cmpq   %rax, %rbx
    jb     .Loop
```

Memory write now unconditional!

Loop Unrolling

- Amortize cost of loop condition by duplicating body
- Creates opportunities for CSE, code motion, scheduling
- Prepares code for vectorization
- Can hurt performance by increasing code size

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

When would this change be incorrect?

Scheduling

- **Rearrange instructions to make it easier for the CPU to keep all functional units busy**
- **For instance, move all the loads to the top of an unrolled loop**
  - Now maybe it’s more obvious why we need lots of registers

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

```
for (size_t i = 0; i < nelts - 4; i += 4) {
    B0 = B[i]; B1 = B[i+1]; B2 = B[i+2]; B3 = B[i+3];
    C0 = C[i]; C1 = C[i+1]; C2 = C[i+2]; C3 = B[i+3];
    A[i  ] = B0*k + C0;
    A[i+1] = B1*k + C1;
    A[i+2] = B2*k + C2;
    A[i+3] = B3*k + C3;
}
```

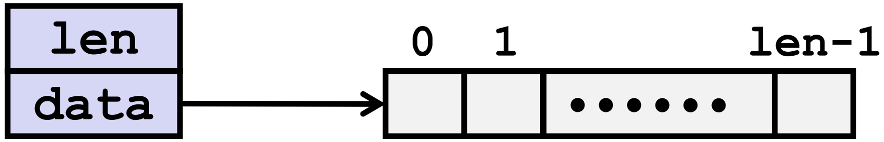
When would *this* change be incorrect?

Today

- Principles and goals of compiler optimization
- Examples of optimizations
- Obstacles to optimization
- Machine-dependent optimization
- **Benchmark example**

Benchmark Example: Data Type for Vectors

```
/* data structure for vectors */
typedef struct{
    size_t len;
    data_t *data;
} vec;
```



- **Data Types**
  - Use different declarations for **`data_t`**
  - `int`
  - `long`
  - `float`
  - `double`

```
/* retrieve vector element
   and store at val */
int get_vec_element
  (*vec v, size_t idx, data_t *val)
{
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}
```

Benchmark Computation

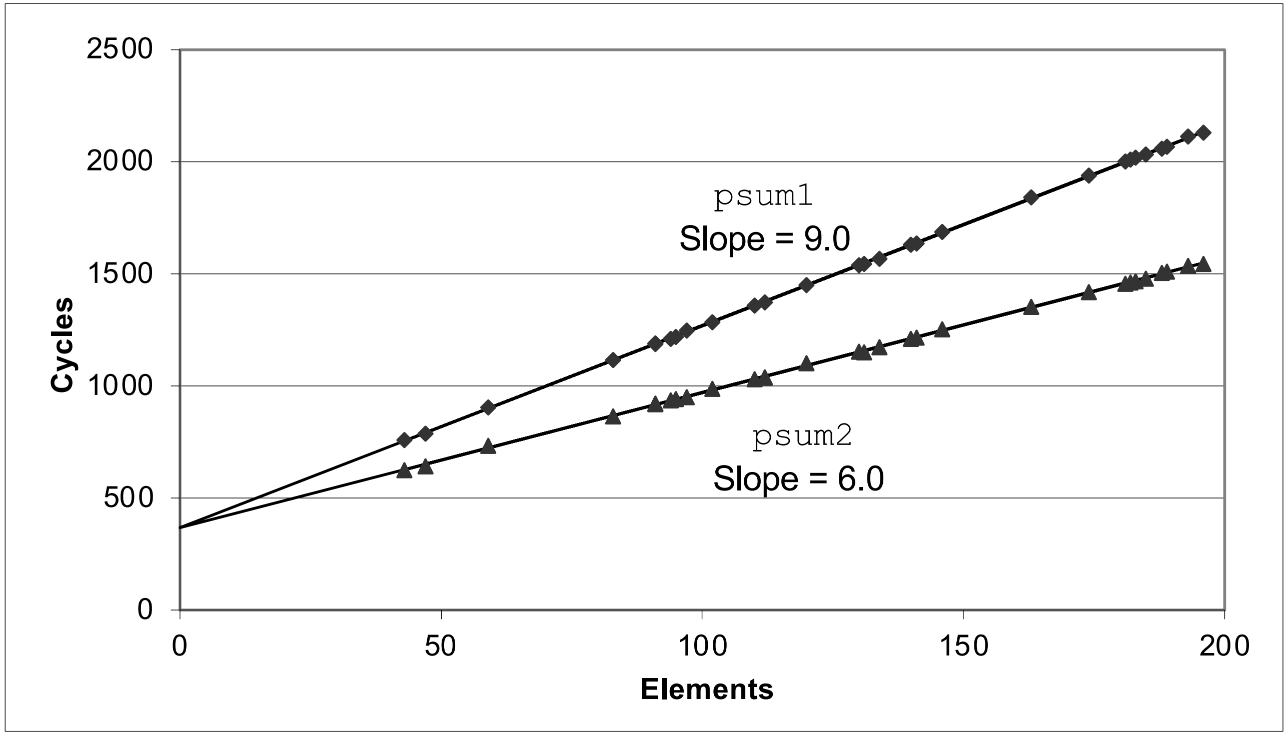
```
void combine1(vec_ptr v, data_t *dest)
{
    long int i;
    *dest = IDENT;
    for (i = 0; i < vec_length(v); i++) {
       data_t val;
       get_vec_element(v, i, &val);
       *dest = *dest OP val;
    }
}
```

Compute sum or product of vector elements

- **Data Types**
  - Use different declarations for `data_t`
  - `int`
  - `long`
  - `float`
  - `double`
- **Operations**
  - Use different definitions of `OP` and `IDENT`
  - `+ / 0`
  - `* / 1`

Cycles Per Element (CPE)

- **Convenient way to express performance of program that operates on vectors or lists**
- **Length = n**
- **In our case: CPE = cycles per OP**
- **Cycles = CPE*n + Overhead**
  - CPE is slope of line



Benchmark Performance

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

Compute sum or product of vector elements

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8

Results in CPE (cycles per element)

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

Effect of Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8
Combine4	1.27	3.01	3.01	5.01

Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x0 = IDENT;
    data_t x1 = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
       x0 = x0 OP d[i];
       x1 = x1 OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x0 = x0 OP d[i];
    }
    *dest = x0 OP x1;
}
```

Loop Unrolled Assembly

- **Remember modern CPU designs**
  - Multiple functional units

- **So how many cycles should this loop take to execute?**

```
.L3:
        imulq   (%rdx), %rcx
        addq    $16, %rdx
        imulq   -8(%rdx), %rdi
        cmpq    %r8, %rdx
        jne     .L3
```

Effect of Loop Unrolling

Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine1 unoptimized	22.68	20.02	19.98	20.18
Combine1 –O1	10.12	10.12	10.17	11.14
Combine1 –O3	4.5	4.5	6	7.8
Combine4	1.27	3.01	3.01	5.01
Unroll	0.81	1.51	1.51	2.51

Multiple instructions every cycle!

Going Further

- **Compiler optimizations are an easy gain**
  - 20 CPE down to 3-5 CPE

- **With careful hand tuning and computer architecture knowledge**
  - 4-16 elements per cycle
  - Newest compilers are closing this gap

- **Use** `gprof`
  - `gcc –Og –pg prog.c –o prog // -pg enables profiling`
  - `./prog file.txt  // generates gmon.out`
  - `gprof prog. // analysis of gmon.out data`

Summary: Getting High Performance

- **Good compiler and flags**
- **Don't do anything sub-optimal**
  - Watch out for hidden algorithmic inefficiencies
  - Write compiler-friendly code
    - Watch out for optimization blockers: procedure calls & memory references
  - Look carefully at innermost loops (where most work is done)

- **Tune code for machine**
  - Exploit instruction-level parallelism
  - Avoid unpredictable branches
  - Make code cache friendly