Do we need more efficiency?

- Some software is fast/small enough
- Some isn't
- More frequent invocations, different work flow
- Bigger inputs
- Better functionality
- Energy savings

Types of efficiency

Run time

- CPU
- hard disk/SSD
- network
- other I/O

Memory

- RAM
- ROM
- persistant storage
- removable storage

Costs of inefficiency

- Loss of user time
- Different work flow
- Misses real time requirements
- More expensive hardware
- Energy

How much efficiency is sensible?

- Command line: 300ms to response
- Music: 20ms latency
- Animated software: screen refresh rate (7-16ms).
- A different component dominates
- Commercial considerations

Other goals

- Correctness
- Simplicity
- Development effort
- Maintenance effort
- Time-to-market
- Security

Extreme positions

- No efficiency considerations
- Optimize everything!

Observations

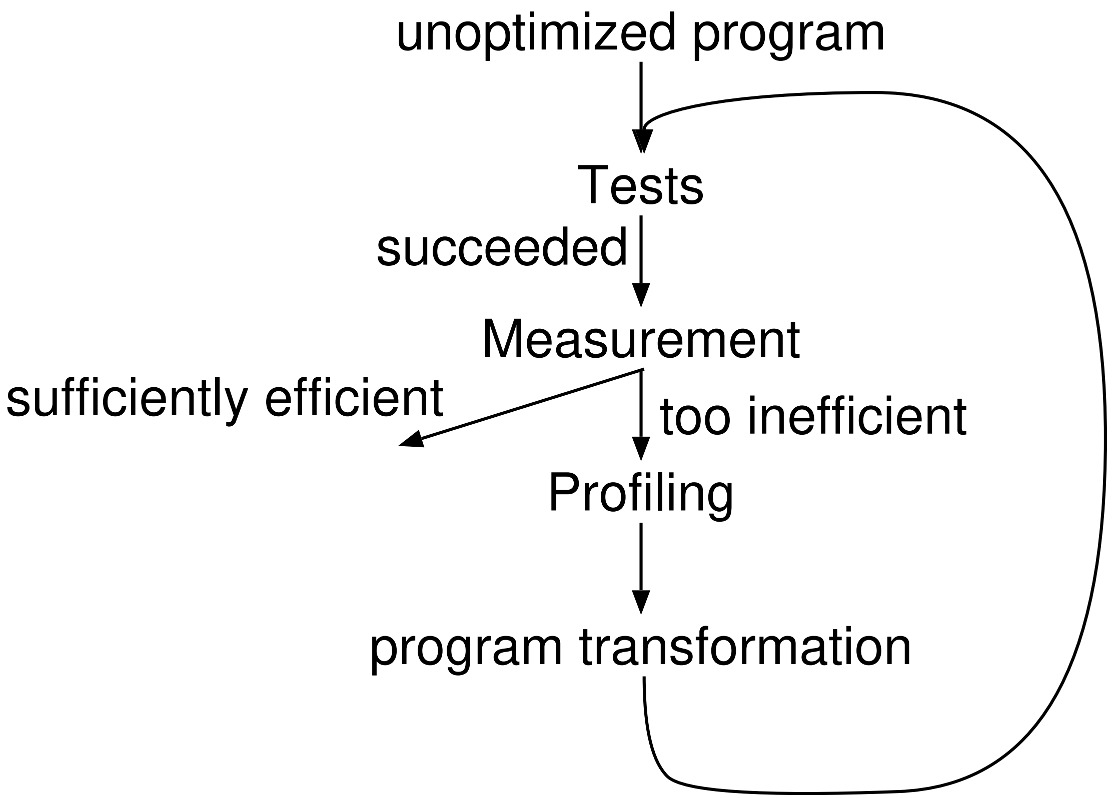
- 80-20 Rule
- Programmers are bad at predicting hot spots

General approach

- Start simple, flexible, maintainable
- Measure
- Optimize critial parts

Problem: Bad efficiency due to specification and design

Method


unoptimized program
Tests
succeeded
Measurement
sufficiently efficient
too inefficient
Profiling
program transformation

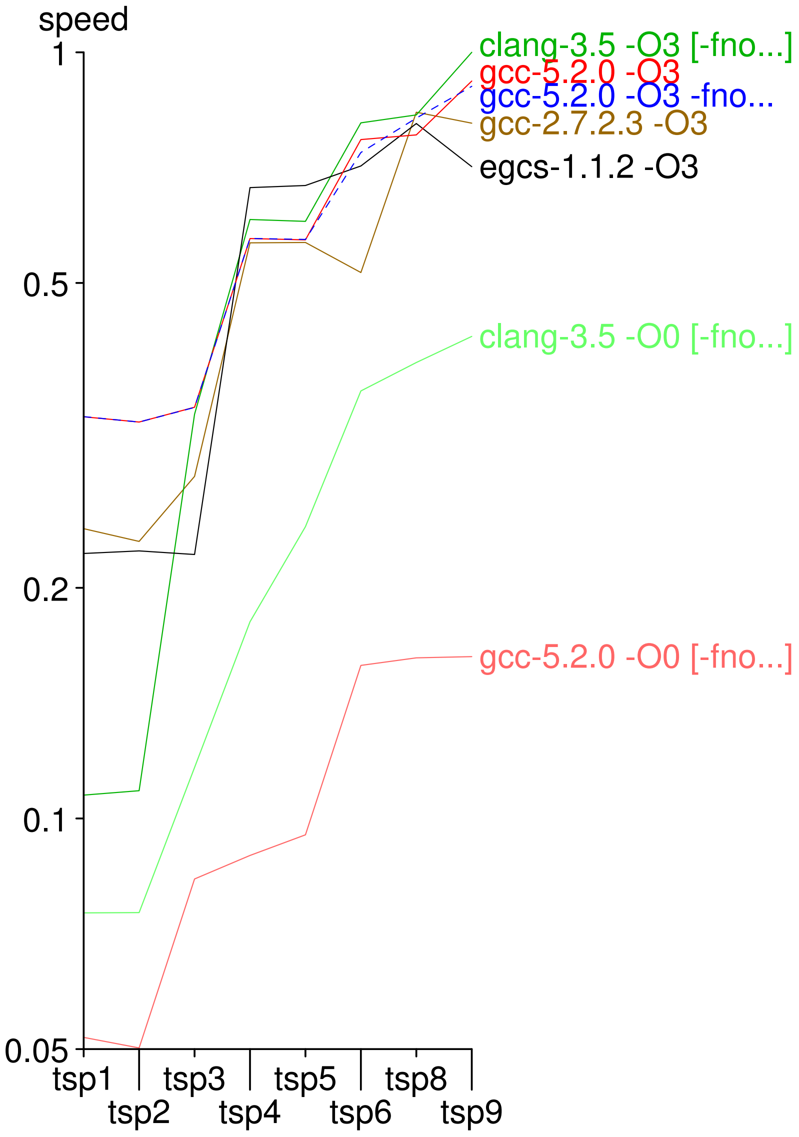
Is this not a job for the compiler?

Compilers use program transformations, too, but

- use the input program as specification
- avoids potential pessimizations
- only performs optimizations that use little time and space during compilation.
- only performs optimizations useful for many applications (or for benchmarks)
- optimizations depend on each other

```
*s1==*s2 && *s1!=0 && *s2!=0
```

Optimization: Compiler vs. Programmer



Example: Stumbling blocks for compilers

```
for (i=0, best=0; i<n; i++)
  if (a[i]<a[best])
    best=i;
return best;

for (p=a, bestp=a, endp=a+n; p<endp; p++)
  if (*p < *bestp)
    bestp = p;
return bestp-a;

for (i=0, bestp=a; a+i<a+n; i++)
  if (a[i]<*bestp)
    bestp=a+i;
return bestp-a;
```

Common stumbling blocks for compilers

- Aliasing

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

- side effects, exceptions

```
if (flag)                   for (i=0; i<n; i++)
  printf(...)                 a[i] = a[i]+1/b[j];
```

Hardware properties

1c	2–8 independent instructions
1c	latency of an ALU instruction
3–5c	latency of a load (L1-hit)
14c	latency of a load (L1-miss, L2-hit)
50c	latency of a load (L2-miss, L3-hit)
50–ns	latency of a load (L3-miss, main memory access)
3ns	Transmission of a cache line (64B) from/to DDR4-2666, DDR5-5200
0–1c	correctly predicted branch
20c	mispredicted branch
4c	latency integer multiply
4c	latency FP addition/multiplication
30–90c	latency division
>100us	IP-Ping in local ethernet Ethernet
10us	1KB transmission across GB Ethernet
10ms	latency hard disk access (seek+rotational delay)
10ms	2500KB sequential hard disk access (without delay)

Hardware properties: latency

```
while (i<n) {
  r+=a[i];
  i++;
}
```

```
add   (%rdi),%rax

add   $0x8,%rdi

cmp   %rdx,%rdi
jne   top1
```

```
while (a!=0) {
  r += a->val;
  a = a->next;
}
```

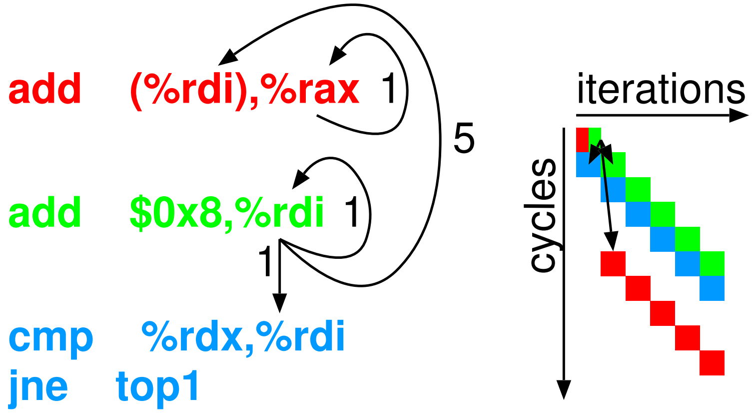
```
add   0x8(%rdi),%rax

mov   (%rdi),%rdi

test  %rdi,%rdi
jne   top2
```

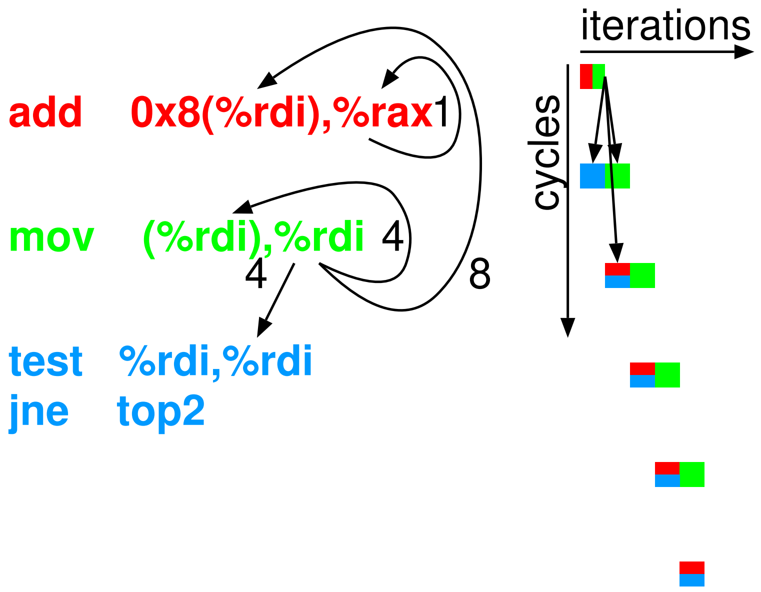
Hardware properties: latency

```
while (i<n) {
  r+=a[i];
  i++;
}
```



Skylake: 1.29c/Iteration

```
while (a!=0) {
  r += a->val;
  a = a->next;
}
```



Skylake: 4c/iteration

Program properties: latency vs. throughput

```
// double a[], r;
while (i<n) {
  r+=a[i];
  i++;
}
```

Skylake: 4c/Iteration

```
// double a[], f;
while (i<n) {
  a[i]=a[i]+f;
  i++;
}
```

Skylake: 1.37c/iteration

with vectorization:

gcc -O3 -mavx:

Skylake: 0.45c/iteration

Program properties

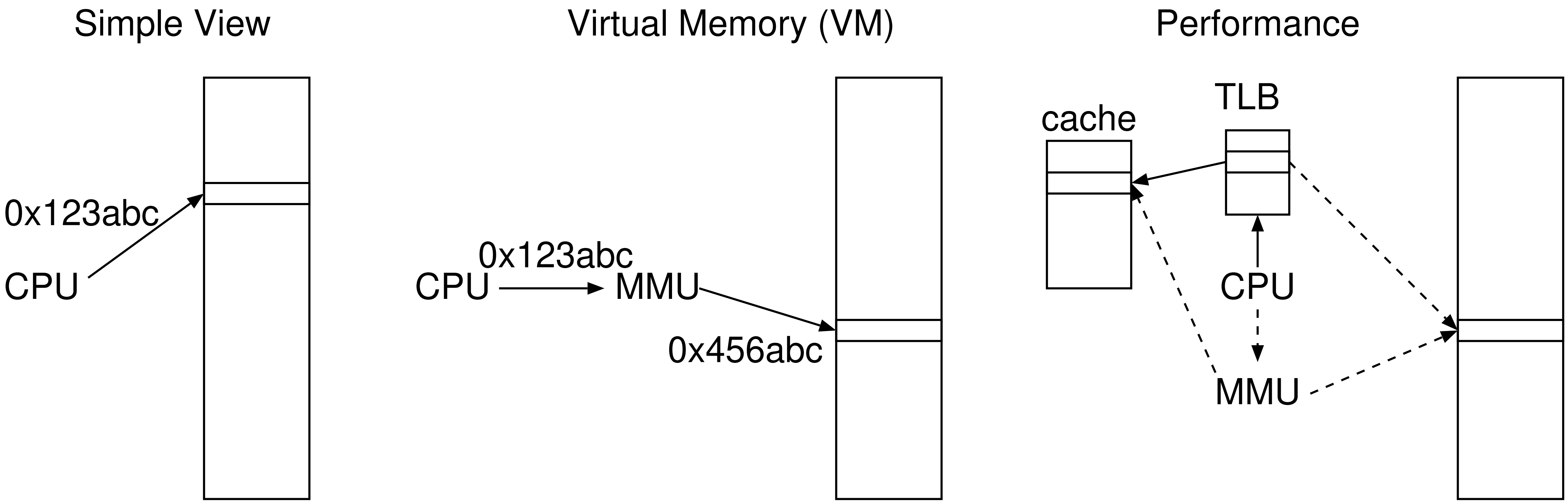
Latency dominated

- depenent operations
  on the same data
- data often is in the cache
- most code (by lines)
- helpful:
  OoO, branch prediction, caches
- sometimes independent instances
  e.g., compilers, on-line-systems
  helpful: multi-core CPUs

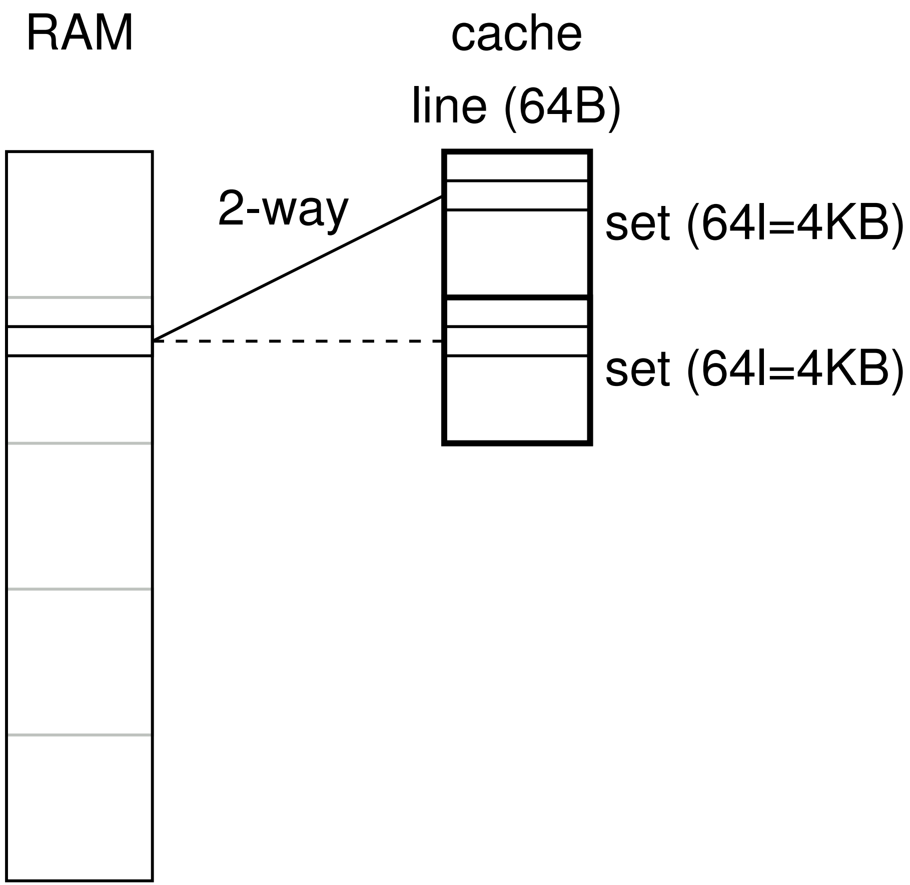
Throughput dominated

- same operations on lots of data
  e.g., pictures, audio, grafics,
  matrices, tensors, neural nets
- often needs (main) memory bandwidth
- little code (by lines)
  much run time
- helpful: SIMD, multi-core CPUs, GPUs
  memory bandwidth

Hardware properties: memory/cache



Hardware properties: memory/cache



- temporal locality (program property)
  spatial locality (program property)
- compulsory misses (program property)
  capacity misses
  conflict misses
- Intel Skylake (Core ix-6xxx):
  data cache (L1): 32KB, 64B/line, 8-way, 4c
  instruction cache (L1): 32KB, 64B/line, 8-way
  L2 cache: 256KB, 64B/line, 4-way, 12c
  L3 cache: 2-8MB, 64B/line, 4-16-way, ≥ 42c
  RAM: ≈ 50ns
  DTLB L1: 64 entries (4KB), 4-way
  DTLB L1: 32 entries (2MB), 4-way
  DTLB L2: 1536 e. (4KB, 2MB), 12-way, 9c

Data structures and algorithms

- Efficient implementation of an inefficient algorithm? Waste of time
- Efficient algorithm, never mind implementation efficiency?
- Efficient implementation of an efficient algorithm
- Efficient algorithm/data structure may conflict with simplicity
- Data structure may affect much of the code
- Abstract data type
  Inefficiency due to abstraction:
  interface overhead
  lack of cost awareness

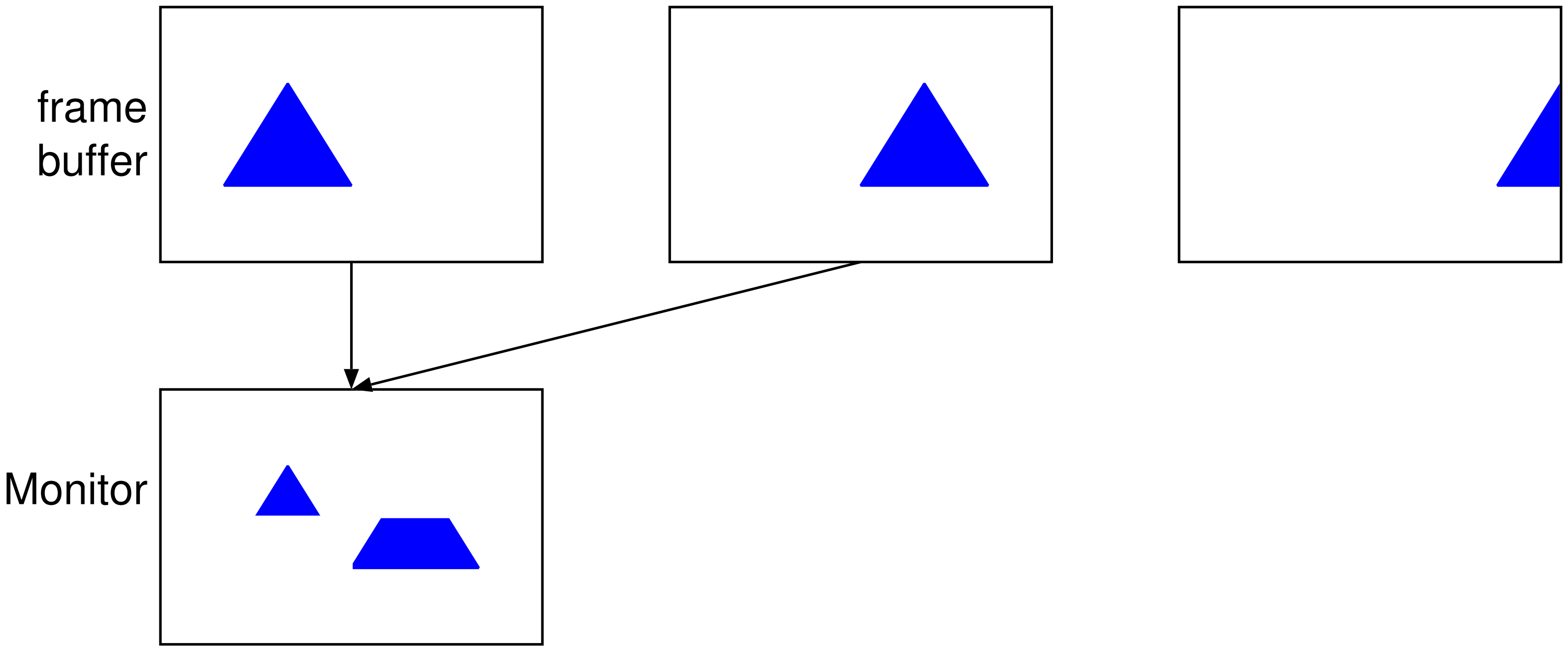
Algorithmic complexity ($O(\ldots)$)

- Helpful, but be aware of its limitations
- Often looks at the worst case
- Counts certain operations, not always relevant for run time
- Ignores constant factors
- logarithmic factors
- E.g.: Search substring (length m) in string (length n)
  simple algorithm: $O(mn)$ (worst), $O(n)$ (best)
  KMP: $O(n)$, but usually slower than the simple algorithm
  BM: $O(n)$ (worst), $O(n/m)$ (best)
- Quicksort: $O(n^2)$ (worst), $O(n \ln n)$ (usual), spatial and temporal locality
  Heapsort: $O(n \ln n)$, bad locality
  Mergesort: $O(n \ln n)$, good locality

Parallel processing

- Problems: find parallelism, express parallelism, synchronization overhead
- Between CPU cores: multithreading, parallel computing
- Between CPU and mass storage: prefetching, write buffering
- Between graphics card and screen: triple buffering
- Between CPU und main memory: prefetching
- Between instructions: instruction scheduling
- SIMD

Triple buffering



- Double buffering without vertical sync: Tearing
- Double buffering with vertical sync: Warten auf vsync
- Triple buffering: no tearing and no waiting

Exploit Word Parallelism/SIMD

```
for (count=0; x > 0; x >>= 1)
  count += x&1;

/* 64-bit-spezifisch */
x = (x & 0x5555555555555555L) + ((x>>1) & 0x5555555555555555L);
x = (x & 0x3333333333333333L) + ((x>>2) & 0x3333333333333333L);
x = (x+(x>>4))   &0x0f0f0f0f0f0f0f0fL;
x = (x+(x>>8)) /*&0x001f001f001f001fL*/;
x = (x+(x>>16))/*&0x0000003f0000003fL*/;
x = (x+(x>>32))  &0x7fL;
count = x;

0|0|0|1|1|0|1|1
  0|  1|  1|  2
      1|      3
              4
```

Efficiency in specification: Copy a memory block

	`cmove` (Forth) `rep movsb` (AMD64)	`memcpy()` (C)	`memmove()` (C) `move` (Forth)
no overlap	source → dest.	source → dest.	source → dest.
start of dest. in source	pattern replication	undefined	source → dest.
start of source in dest.	source → dest.	undefined	source → dest.
implementation	byte by byte	bigger units	decision
efficient implementation	decision		
	overspecified	underspecified	well specified

Programming languages

- inherent inefficiency
- idiomatic inefficiency
- compiler efficiency
- (potential) efficiency due to development speed
- assembly language?

Programming languages: Examples

- Aliasing: C vs. Fortran (inherent)

```
void f(double a[], double b[], double c[], long n) {
  for (long i=0; i<n; i++)
    c[i]=a[i]+b[i];
}
```

Programming languages: Examples

- Nested data: Java vs. C(++) (inherent)

```
struct mystruct { int a; float b; double c; }
struct mystruct a[10000];
struct mystruct *b[10000];
```

- Scaling in address arithmetics: C vs. Forth (inherent/idiomatic)

```
mystruct *p;              ... constant p
mystruct *q;              ... constant q
...                       ...
long d = q-p;             q p - constant d1
mystruct *r = p+d;        p d1 + constant r
```

Programming languages: examples

- 0-terminated strings in C (inherent/idiomatic)

```
l=strlen(s);
strcat(strcat(strcat(s,s1),s2),s3);
```

- “C++ ist slow” (idiomatic)
- Microbenchmarks (compiler)
- programming contests (development speed)
- Riad air port

Code motion out of loops

```
for (...) {
  .... computation ...
}
```

computation has no side effects
computation does not need values computed in the loop

```
temp = computation;
for (...) {
  .... temp ...
}
```

Combining Tests

E.g., sentinel in search loops

```
for (i=0;  i<n && a[i]!=key; i++)
```

`a[n]` is writable

```
a[n] = key;
for (i=0; a[i]!=key; i++)
    ;
```

lowers maintainability, reentrancy

Loop Unrolling

```
for (i=0; i<n; i++)
  body(i);
```

```
for (i=0; i<n-1; i+=2) {
  body(i);
  body(i+1);
}
for (; i<n; i++)
  body(i);
```

Transfer-Driven Unrolling/Modulo Variable Renaming

```
new_a = ...
... = ... a ...
a = new_a
```

Unrolling by 2

```
a2 = ...;
... = ... a1 ...;
a1 = ...;
... = ... a2 ...;
```

Software Pipelining

```
for (...) {
  a = ...;
  ... = ... a ...;
}
```

Computing `a` has no side effects

```
a = ...;
for (...) {
  ... = ... a ...;
  a = ...;
}
```

```
new_a = ...;
for (...) {
  a = new_a;
  new_a = ...;
  ... = ... a ...;
}
```

Unconditional Branch Removal

```
while (test)
  code;
```

```
if (test)
  do
    code;
  while (test);
```

Loop Peeling

```
while (test)
  code;
```

```
if (test) {
  code;
  while (test)
    code;
}
```

Loop Fusion

```
for (i=0; i<n; i++)
  code1;
for (i=0; i<n; i++)
  code2;
```

Iteration k in `code2` does not depend on Iteration $j > k$ in `code1`.
`Code2` does not overwrite data that is read by `code1`.

```
for (i=0; i<n; i++) {
  code1;
  code2;
}
```

Exploit Algebraic Identities

`~a&~b`

`~(a|b)`

Computer “integers” are not $\mathbb{Z}$.
FP numbers are not $\mathbb{R}$.

Integer: Overflow: $a > b \not\Leftrightarrow a + n > b + n$

FP: Rundungsfehler: $a + (b + c) \neq (a + b) + c$

Short-circuiting Monotone Functions

```
for (i=0, sum=0; i<n; i++)
  sum += x[i];
flag = sum > cutoff;
```

All `x[i]>=0`, `sum` and `i` are not used later.

```
for (i=0, sum=0; i<n && sum <= cutoff; i++)
  sum += x[i];
flag = sum > cutoff;
```

Unrolling for fewer comparisons and branches.

Arithmetics with flags

```
if (flag)
  x++;
```

```
x += (flag != 0);
```

Different representation of flags

```
(a<0) != (b<0)
```

```
(a^b) < 0
```

Long-circuiting

`A && B`

A and B compute flags, B has no side effects

`A & B`

When to use: If B is cheap and A is hard to predict

Reordering Tests

`A && B`

A and B have no side effects

`B && A`

Which order?

- Cheaper first
- More predictable first
- higher probability of short-circuiting first

Reordering Tests

```
if (A)
  ...
else if (B)
  ...
```

A and B have no side effects, $\neg(A \wedge B)$.

```
if (B)
  ...
else if (A)
  ...
```

Boolean/State Variable Elimination

```
flag = ...;
S1;
if (flag)
  S2;
else
  S3;
```

`flag` is not used later.

```
if (...) {
  S1;
  S2;
} else {
  S1;
  S3;
}
```

Collapsing Procedure Hierarchies

- Inlining
- Specialization

```
foo(int i, int j)
{
...
}
... foo(1, a);
```

```
foo_1(int j)
{
  ...
}
```

Precompute Functions

```
int foo(char c)
{
    ...
}
```

foo() has no side effects.

```
int foo_table[] = {...};

int foo(char c)
{
  return foo_table[c];
}
```

Exploit Common Cases

Handle all cases correctly and common cases efficiently.

- Memoization: Remember results of earlier evaluations of an expensive function
- Pre-computed tables or special code sequences for frequent parameters

Coroutines

Instead of multi-pass processing:

```
coroutine producer {
  for (...)
    ... consumer(x); ...
}

coroutine consumer {
  for (...)
    ... x = producer(); ...
}
```

Related: Pipelines, Iterators, etc.

Transformation on Recursive Procedures

- Tail call optimization
- Inlining
- Replace one recursive call by counter
- General case: Use explicit stack
- Use different method for small problems
- Use recursion instead of iteration for automatic cache blocking

Tail Call Optimization

```
void traverse_simple( PNODE p )
{
  if ( p!=0 )
  {
    traverse_simple( p->l );
    ...
    traverse_simple( p->r );
  }
}
```

```
void traverse_simple( PNODE p )
{ start:
  if ( p!=0 )
  {
    traverse_simple( p->l );
    ...
    p = p->r; goto start;
  }
}
```

Replace one recursive call by counter

```
foo()
{
  if (...) {
    code1;
    foo();
    code2;
  }
}
```

```
while (...) {
  count++;
  code1;
}
for (i=0; i<count; i++)
  code2;
```

Compile-Time Initialization

- Initialize tables at compile-time instead of at run-time
- CPU time vs. load time from disk

Strength Reduction/Incremental Algorithms/Differentiation

```
y = x*x;
x += 1;
y = x*x;
```

```
y = x*x;
x += 1;
y += 2*x-1;
```

Common subexpression elimination/Partial Redundancy Elimination

```
a = Exp;
b = Exp;
```

Exp has no side effects

```
a = Exp;
b = a;
```

Pairing Computation

- Additional result for small effort
- E.g., division and remainder (C: `div`)
  sin and cos (glibc: `sincos`)

Data Structure Augmentation

- Redundant data for accelerating certain operations
- Redundancy: possibility of inconsistency
- Caching
- Memoization
- Hints that can be correct, or not (e.g., branch prediction)
- Example: dictionary in Gforth: linked list augmented with hash table

Automata

- state represents something more complex
- finite state machine for scanning
- pushdown automaton for parsing
- tree automaton for instruction selection
  `iburg` (not an automaton) → **`burg`**

Lazy Evaluation

- Example: automaton for regular expressions
- Example: tree-parsing automaton