CSCI 2021: Program Performance
Micro-Optimizations

Chris Kauffman

Last Updated:
Tue Apr 16 09:25:06 CDT 2019

Logistics

Date	Event
Fri 4/12	Lec: Micro-opts
Mon 4/15	Lec: Micro-opts,
Tue 4/16	**A4 DUE**
Wed 4/17	Lec/Lab11: Review
Fri 4/19	Exam 3

Reading Bryant/O'Hallaron

Ch 5: Optimizing Program Performance

Goals

- What to Optimize First
- First-best Optimzations
- Micro-optimization Techniques

Assignment 4

- Questions?

Lab 10

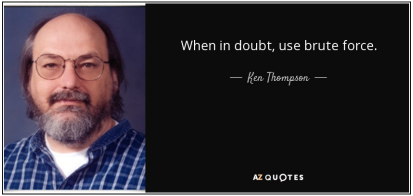
- Comparing micro-optimizations
- Memory optimizations

Caution: Should I Optimize?

- Optimizing program execution time usually costs human time
- Human time is valuable, don't waste it
- Determine if there is a NEED to optimize
- **Benchmark** your code - if it is fast enough, move on
- If not fast enough, use a **profiler** to determine where your efforts are best spent
- **Never sacrifice correctness** for speed

First make it work,
then make it right,
then make it fast.

- Kent Beck

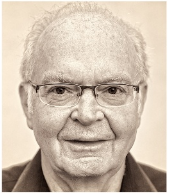


What to Optimize First

In order of impact

1. Algorithms and Data Structure Selection
2. Elimination of unneeded work/hidden costs
3. Memory Utilization
4. **Micro-optimizations**

“Premature optimization is the root of all evil” - Donald Knuth



Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: *premature optimization is the root of all evil.* **Yet we should not pass up our opportunities in that critical 3%.**
– Donald Knuth

Exercise: Optimize This

- Prema Turopt is tasked by her boss to optimize the performance function `get_min()`
- The current version of the function code looks like the code to the right.
- Prema immediately jumps to the code for `bubble_sort()` and alters the code to enable better processor pipelining.
- This leads to a 3% improvement in speed.

```
int get_min(storage_t *st){
  int *arr =
   malloc(sizeof(int)*get_size(st));

  for(int i=0; i<get_size(st); i++){
    arr[i] = get_element(st,i);
  }

  bubble_sort(arr, get_size(st));

  int ans = arr[0];
  free(arr);
  return ans;
}
```

Suggest several alternatives that Prema should have explored

Answers: Optimize This

1. Don't use bubblesort: $O(N^2)$. Use an $O(N \log N)$ sort like Quicksort, Heapsort, Mergesort
2. Why sort at all? Determine the minimum element with the "get" loop.
3. What is the cost of `get_element()` and `get_size()`? Is there a more efficient iterator or array-extraction mechanism?
4. What data structure is used in `storage_t`? If it is already sorted such as a binary search tree or binary heap, there may be a more efficient way to determine the minimum element.

```
int get_min(storage_t *st){
  int *arr =
   malloc(sizeof(int)*get_size(st));

  for(int i=0; i<get_size(st); i++){
    arr[i] = get_element(st,i);
  }

  bubble_sort(arr, get_size(st));

  int ans = arr[0];
  free(arr);
  return ans;
}
```

5. Might be able to arrange for minimum elements to be tracked automatically in the data structure for `storage_t` if other code can be modified - `O(1)` `get_min()` method.

Exercise: Eliminating Unnecessary Work

```
void lower1(char *s) {

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

```
void lower2(char *s) {
  long len = strlen(s);
  for (long i=0; i < len; i++){
    if (s[i] >= 'A' && s[i] <= 'Z'){
      s[i] -= ('A' - 'a');
    }
  }
}
```

- Bryant/O'Hallaron Figure 5.7
- Two versions of a lower-casing function
- Lowercase by subtracting off constant for uppercase characters: alters ASCII code
- Examine them to determine differences
- Project speed differences and **why one will be faster**

Answers: Eliminating Unnecessary Work

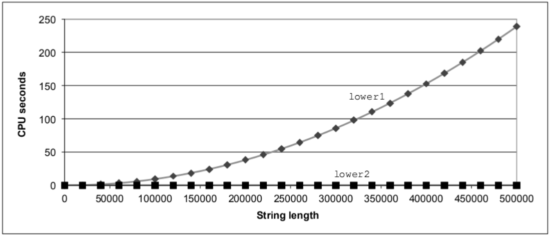
- `strlen()` is $O(N)$: searches for \0 character in `for()` loop
- Don't loop with it if possible

```
void lower1(char *s) {

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

```
void lower2(char *s) {
  long len = strlen(s);
  for (long i=0; i < len; i++){
    if (s[i] >= 'A' && s[i] <= 'Z'){
      s[i] -= ('A' - 'a');
    }
  }
}
```

```
long strlen(char *s) {
  long len = 0;
  while(s[len] != '\0'){
    len++;
  }
  return len;
}
```



Exercise: Allocation and Hidden Costs

Consider the following **Java** code

```
public class StringUtils{
  public static
  String repString(String str, int reps)
  {
    String result = "";
    for(int i=0; i<reps; i++){
      result = result + str;
    }
    return result;
  }
}
```

- Give a Big-O estimate for the runtime
- Give a Big-O estimate for the memory overhead

Answers: Allocation and Hidden Costs

- Strings are **immutable** in Java (Python, many others)
- Each iteration must
  - **allocate** new memory for a new string sized
    `result.length + str.length`
  - Copy `result` to the first part
  - Copy `str` to the second part
- Leads to $O(N^2)$ complexity
- Much worse memory usage: as much as $O(N^2)$ wasted memory for garbage collector to clean up

```
public class StringUtils{
  public static
  String repString(String str, int reps)
  {
    String result = "";
    for(int i=0; i<reps; i++){
      result = result + str;
    }
    return result;
  }

  // Efficient version
  public static
  String repString2(String str, int reps)
  {
    StringBuilder result =
      new StringBuilder();
    for(int i=0; i<reps; i++){
      result.append(str);
    }
    return result.toString();
  }

}
```

Exercise: Do Memory References Matter?

```
void sum_range1(int start,
                int stop,
                int *ans)
{
  *ans = 0;
  for(int i=start; i<stop; i++){
    *ans += i;
  }

}
```

```
void sum_range2(int start,
                int stop,
                int *ans)
{
  int sum = 0;
  for(int i=start; i<stop; i++){
    sum += i;
  }
  *ans = sum;
}
```

- What is the primary difference between the two routines above?
- What effect if any will this have on runtime?

Answers: Do Memory References Matter?

```
void sum_range1(int start,
                int stop,
                int *ans)
{
  *ans = 0;
  for(int i=start; i<stop; i++){
    *ans += i;
  }

}
```

`sum_range1()` makes repeated memory references

```
void sum_range2(int start,
                int stop,
                int *ans)
{
  int sum = 0;
  for(int i=start; i<stop; i++){
    sum += i;
  }
  *ans = sum;
}
```

`sum_range2()` uses a local variable with only a couple memory references

- Must determine if memory references matter for performance
- Guesses?

Memory References Matter, Compiler May Change Them

```
lila> gcc -Og sum_range.c   # No opt
lila> ./a.out 0 1000000000
sum_range1: 1.9126e+00 secs
sum_range2: 2.6942e-01 secs
```

- Minimal optimizations
- Memory reference definitely matters

```
lila> gcc -O1 sum_range.c  # Opt plz
lila> ./a.out 0 1000000000
sum_range1: 2.8972e-01 secs
sum_range2: 2.7569e-01 secs
```

- Observe code differences between -Og and -O1
- Why is performance improved so much?

```
### Compiled with -Og: minimal opt
sum_range1:
  movl     $0, (%rdx)    # write to memory
  jmp      .LOOTOP
.BODY:
  addl     %edi, (%rdx) # memory write
  addl     $1, %edi      # in loop
.LOOPTOP:
  cmpl     %esi, %edi
  jl       .BODY
  ret

### Compiled with -O1: some opt
sum_range1:
  movl     $0, (%rdx)    # mem write
  cmpl     %esi, %edi
  jge      .END
  movl     $0, %eax      # 0 to reg
.LOOP:
  addl     %edi, %eax    # add to reg
  addl     $1, %edi      # no mem ref
  cmpl     %edi, %esi
  jne      .LOOP
  movl     %eax, (%rdx) # write at end
.END:
  ret
```

Dash-O! Compiler Optimizes for You

- `gcc` like, many compilers, can perform some memory and micro-optimizations for you
- Series of `-OX` options to enable different techniques
- We will use `-Og` at times to disable many optimizations
    - `-Og`: Optimize debugging. … optimization level of choice for the standard edit-compile- debug cycle
- Individual optimizations can be enabled and disabled

- `-O` or `-O1`: Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.
  With -O, the compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.
- `-O2`: Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed tradeoff. As compared to `-O`, this option increases both compilation time and the performance of the generated code.
- `-O3`: Optimize yet more. `-O3` turns on all optimizations specified by -O2 and also…
- `-Ofast`: Disregard strict standards compliance. (!)

Compiler Optimizations

-O turns on the following optimization flags:

```
-fauto-inc-dec -fbranch-count-reg -fcombine-stack-adjustments
--fcompare-elim fcprop-registers -fdce -fdefer-pop -fdelayed-branch
--fdse -fforward-propagate fguess-branch-probability -fif-conversion2
--fif-conversion finline-functions-called-once -fipa-pure-const
--fipa-profile -fipa-reference fmerge-constants -fmove-loop-invariants
--freorder-blocks -fshrink-wrap fshrink-wrap-separate
--fsplit-wide-types -fssa-backprop -fssa-phiopt -ftree-bit-ccp
-ftree-ccp -ftree-ch -ftree-coalesce-vars -ftree-copy-prop -ftree-dce
-ftree-dominator-opts -ftree-dse -ftree-forwprop -ftree-fre
--ftree-phiprop -ftree-sink ftree-slsr -ftree-sra -ftree-pta
--ftree-ter -funit-at-a-time
```

- Some combination of these enables `sum_range2()` to fly as fast as `sum_range1()`
- We will look at some "by-hand" versions of these optimizations but whenever possible, let the compiler do it

Exercise: Loop Unrolling

- Have seen copying loop iterations manually *may* lead to speed gains
- **Why?** Which of the following unrolled versions of `sum_rangeX()` seems fastest?
- Why the **second loop** in `sum_rangeB()` and `sum_rangeC()`?

```
void sum_rangeA(long stop, long *ans){
  long sum=0, i;
  for(i=0; i<stop; i++){
    sum += i+0;
  }
  *ans = sum;
}

```

```
void sum_rangeB(long stop, long *ans){
  long sum = 0, i;
  for(i=0; i<stop-3; i+=3){
    sum += (i+0);
    sum += (i+1);
    sum += (i+2);
  }
  for(; i<stop; i++){
    sum += i;
  }
  *ans = sum;
}

void sum_rangeC(long stop, long *ans){
  long sum0=0, sum1=0, sum2=0, i;
  for(i=0; i<stop-3; i+=3){
    sum0 += (i+0);
    sum1 += (i+1);
    sum2 += (i+2);
  }
  for(; i<stop; i++){
    sum0 += i;
  }
  *ans = sum0 + sum1 + sum2;
}
```

Exercise: Loop Unrolling

Expectations

Version	Notes	Performance
`sum_rangeA()`	Not unrolled	Baseline
`sum_rangeB()`	Unroll x3, same destinations for sum	Less good
`sum_rangeC()`	Unroll x3, different destinations sum add	Expected Best

Actual Performance

```
apollo> gcc -Og unroll.c
apollo> ./a.out 1000000000
sum_rangeA: 1.0698e+00 secs
sum_rangeB: 6.2750e-01 secs
sum_rangeC: 6.2746e-01 secs

phaedrus> ./a.out 1000000000
sum_rangeA: 2.8913e-01 secs
sum_rangeB: 5.3285e-01 secs
sum_rangeC: 2.6774e-01 secs
```

Unrolling is Unpredictable

- Performance Gains vary from one compiler+processor to another
- All unrolling requires **cleanup loops** like those in the B/C versions: add on remaining elements

GCC Options to Unroll

- gcc has options to unroll loops during optimization
- Unrolling has unpredictable performance implications so unrolling is **not enabled** for `-O1, -O2, -O3`
- Can manually enable it with compiler options like `-funroll-loops` to check for performance bumps

```
apollo> gcc -Og unroll.c
apollo> ./a.out 1000000000
sum_rangeA: 1.0698e+00 secs
sum_rangeB: 6.2750e-01 secs
sum_rangeC: 6.2746e-01 secs
```

```
apollo> gcc -O3 unroll.c
apollo> ./a.out 1000000000
sum_rangeA: 9.4124e-01 secs
sum_rangeB: 4.1833e-01 secs
sum_rangeC: 4.1832e-01 secs
# manual unroll + compiler opt
```

```
apollo> gcc -Og -funroll-loops unroll.c
apollo> ./a.out 1000000000
sum_rangeA: 7.0386e-01 secs    # loop unrolled by compiler
sum_rangeB: 6.2802e-01 secs
sum_rangeC: 6.2797e-01 secs

apollo> gcc -Og -funroll-loops -fvariable-expansion-in-unroller unroll.c
apollo> ./a.out 1000000000
sum_rangeA: 5.2711e-01 secs    # unroll + multiple intermediates used
sum_rangeB: 6.2759e-01 secs
sum_rangeC: 6.2750e-01 secs
```

Do Conditionals Matter?

Consider two examples of adding even numbers in a range

```c
// CONDITION version
long sum_evensA(long start, long stop){
  long sum=0;
  for(int i=start; i<stop; i++){
    if((i & 0x01) == 0){
      sum += i;
    }
  }
  return sum;
}
// STRAIGHT-LINE version
long sum_evensB(long start, long stop){
  long sum=0;
  for(int i=start; i<stop; i++){
    int odd = i & 0x01;
    int even_mask = odd - 1;
    // 0x00000000 for odd
    // 0xFFFFFFFF for even
    sum += even_mask & i;
  }
  return sum;
}
```

Timings for these two are shown below at two levels of optimization.

```
lila> gcc -Og condloop.c
lila> a.out 0 400000000
sum_evensA: 1.1969e+00 secs
sum_evensB: 2.8953e-01 secs
# 4x speedup

lila> gcc -O3 condloop.c
lila> a.out 0 400000000
sum_evensA: 2.3662e-01 secs
sum_evensB: 9.6242e-02 secs
# 2x speedup
```

Message is simple: **eliminate conditionals** whenever possible to improve performance

Exercise: Row Sums with Function v Macro

What is the difference between these two row sum functions?

```c
int mget(matrix_t mat,
         int i, int j)
{
  return
    mat.data[i*mat.cols + j];
}
int vset(vector_t vec,
         int i, int x)
{
  return vec.data[i] = x;
}
void row_sumsA(matrix_t mat,
               vector_t sums)
{
  for(int i=0; i<mat.rows; i++){
    int sum = 0;
    for(int j=0; j<mat.cols; j++){
      sum += mget(mat,i,j);
    }
    vset(sums, i, sum);
  }
}
```

```c
#define MGET(mat,i,j) \
  ((mat).data[((i)*((mat).cols)) + (j)

#define VSET(vec,i,x) \
  ((vec).data[(i)] = (x))

void row_sumsB(matrix_t mat,
               vector_t sums)
{
  for(int i=0; i<mat.rows; i++){
    int sum = 0;
    for(int j=0; j<mat.cols; j++){
      sum += MGET(mat,i,j);
    }
    VSET(sums, i, sum);
  }
}
```

Answers: Row Sums with Function v Macro

- `row_sumsA()` uses standard function calls to retrieve elements
- `row_sumsB()` uses **macros** to do the element retrieval
- A macro is a textural expansion done by the **preprocessor**: insert the literal text associated with the macro
- See macro results with

  ```
  gcc -E rowsums.c
  ```

  which stops after preprocessor step (early)

- Function calls cost some operations but not many
- Function calls **prevent optimization across boundaries**
- Cannot pipeline effectively when jumping around, using registers for arguments, restoring registers, etc
- Macros can alleviate this but they are a **pain** to write and notoriously buggy
- Better to let the compiler do this for us

Inlining Functions/Procedures

- **Function Inlining** inserts the body of a function where it would have been called
- Turned on fully partially at -O2 and fully at -O3
- Enables other optimizations blocked by function boundaries
- Can only be done if source code (C file) for function is available
- Like loop unrolling, function inlining has trade-offs
  - Enables pipelining
  - More predictable control
  - More register pressure
  - Increased code size

```
> FILES="rowsums.c matvec_util.c"
> gcc -Og $FILES
> ./a.out 8000 8000
row_sumsA: 1.3109e-01 secs
row_sumsB: 4.0536e-02 secs

> gcc -Og -finline-small-functions $FILES
> ./a.out 8000 8000
row_sumsA: 7.4349e-02 secs
row_sumsB: 4.2682e-02 secs

> gcc -O3 $FILES
> ./a.out 8000 8000
row_sumsA: 2.1974e-02 secs
row_sumsB: 2.0820e-02 secs
```

- Inlining typically most effective for for small functions (getters/setters)

Profilers: gprof and Friends

- **Profiler**: a tool that monitors code execution to enable performance optimizations
- gprof is stock on Linux systems, interfaces with gcc
- Compile with profiling options: gcc -pg
- Run code to produce data file
- Examine with gprof
- **Note:** gcc version 6 and 7 contain a bug requiring use of -no-pie option, not a problem on apollo

```
# Compile
# -pg : instrument code for profiling
# -no-pie : bug fix for new-ish gcc's
> gcc -pg -no-pie -g -Og -o unroll unroll.c

> ls
unroll  unroll.c

> ./unroll 1000000000
sum_rangeA: 2.9401e-01 secs
sum_rangeB: 5.3164e-01 secs
sum_rangeC: 2.6574e-01 secs

# gmon.out now created with timing info
> ls
gmon.out  unroll  unroll.c

> file gmon.out
gmon.out: GNU prof performance data

> gprof -b unroll
... output on next slide ...
```

gprof output for unroll

```
> gprof -b unroll
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.38      0.54     0.54        1   544.06   544.06  sum_rangeB
 26.12      0.83     0.28        1   282.11   282.11  sum_rangeA
 24.26      1.09     0.26        1   261.95   261.95  sum_rangeC

                        Call graph
index % time    self  children    called     name
[1]    100.0    0.00    1.09                 main [1]
                0.54    0.00       1/1           sum_rangeB [2]
                0.28    0.00       1/1           sum_rangeA [3]
                0.26    0.00       1/1           sum_rangeC [4]
-----------------------------------------------
                0.54    0.00       1/1           main [1]
[2]     50.0    0.54    0.00       1         sum_rangeB [2]
-----------------------------------------------
                0.28    0.00       1/1           main [1]
[3]     25.9    0.28    0.00       1         sum_rangeA [3]
-----------------------------------------------
                0.26    0.00       1/1           main [1]
[4]     24.1    0.26    0.00       1         sum_rangeC [4]
-----------------------------------------------
```

gprof Example: Dictionary Application

```
> ./dictionary < craft-67.txt
... Total time = 0.829561 seconds
> gprof -b dictionary
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.07      0.18     0.18        1   180.25   180.25  sort_words
 19.47      0.25     0.07   463016     0.00     0.00  find_ele_rec
 13.91      0.30     0.05  2862749     0.00     0.00  Strlen
  8.34      0.33     0.03   463016     0.00     0.00  lower1
  2.78      0.34     0.01   463017     0.00     0.00  get_token
  2.78      0.35     0.01   463016     0.00     0.00  h_mod
  2.78      0.36     0.01    20451     0.00     0.00  save_string
  0.00      0.36     0.00   463017     0.00     0.00  get_word
  0.00      0.36     0.00   463016     0.00     0.00  insert_string
  0.00      0.36     0.00    20451     0.00     0.00  new_ele
  0.00      0.36     0.00        7     0.00     0.00  add_int_option
  0.00      0.36     0.00        1     0.00     0.00  add_string_option
  0.00      0.36     0.00        1     0.00     0.00  init_token
  0.00      0.36     0.00        1     0.00     0.00  new_table
  0.00      0.36     0.00        1     0.00     0.00  parse_options
  0.00      0.36     0.00        1     0.00     0.00  show_options
  0.00      0.36     0.00        1     0.00   360.50  word_freq
```

gprof Example Cont'd: Dictionary Application

```
> ./dictionary < craft-67.txt     ## After upgrading sort_words() to qsort()
... Total time = 0.624172 seconds
> gprof -b dictionary
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 60.08      0.12     0.12   463016     0.00     0.00  find_ele_rec
 15.02      0.15     0.03  2862749     0.00     0.00  Strlen
 10.01      0.17     0.02   463016     0.00     0.00  lower1
  5.01      0.18     0.01   463017     0.00     0.00  get_token
  5.01      0.19     0.01   463016     0.00     0.00  h_mod
  5.01      0.20     0.01    20451     0.00     0.00  save_string
  0.00      0.20     0.00   463017     0.00     0.00  get_word
  0.00      0.20     0.00   463016     0.00     0.00  insert_string
  0.00      0.20     0.00    20451     0.00     0.00  new_ele
  0.00      0.20     0.00        8     0.00     0.00  match_length
  0.00      0.20     0.00        7     0.00     0.00  add_int_option
  0.00      0.20     0.00        1     0.00     0.00  add_string_option
  0.00      0.20     0.00        1     0.00     0.00  find_option
  0.00      0.20     0.00        1     0.00     0.00  init_token
  0.00      0.20     0.00        1     0.00     0.00  new_table
  0.00      0.20     0.00        1     0.00     0.00  parse_options
  0.00      0.20     0.00        1     0.00     0.00  show_options
  0.00      0.20     0.00        1     0.00     0.00  sort_words ** was 0.18 **
  0.00      0.20     0.00        1     0.00   200.28  word_freq
```