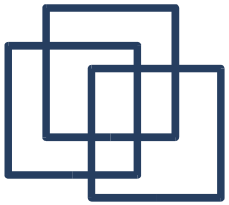
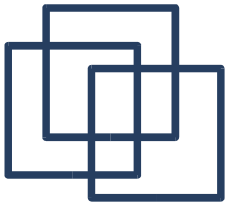


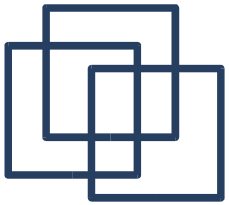
Code Optimization

Emmanuel Fleury
B1-201
fleury@cs.aau.dk

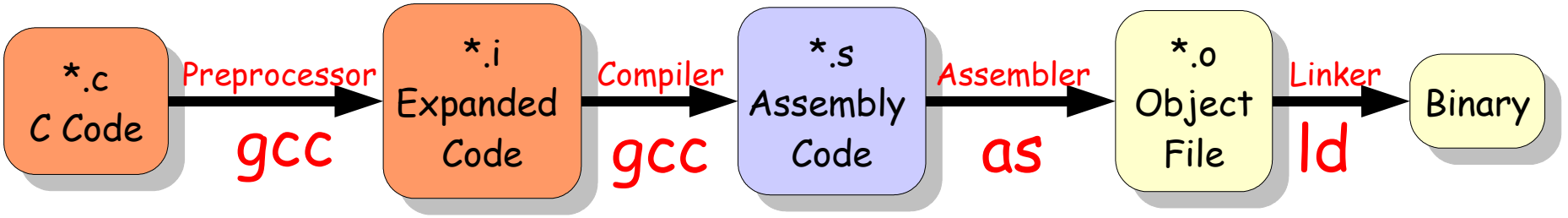
GNU FDL
FREE DOC
LICENSE



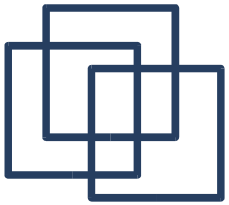
Optimization Options in gcc



Compilation Process



- Preprocessing (to expand macros)
- Compilation (from source code to assembly language)
- Assembly (from assembly language to machine code)
- Linking (to create the final executable)

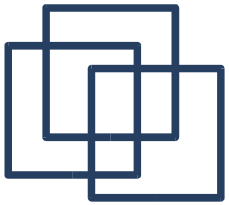


Preprocessing

```
#include <stdio.h>

#define MESSAGE "Hello, world!\n"

int main () {
  printf (MESSAGE);
  return 0;
}
```

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "hello.c"

... [snip] ...

extern void funlockfile (FILE *__stream) ;
# 831 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

int main () {
  printf ("Hello, world!\n");
  return 0;
}
```

Command: `gcc -E hello.c > hello.i`

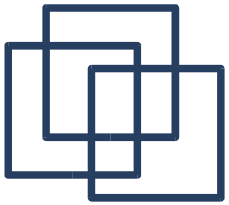


Compilation

```
#include <stdio.h>

#define MESSAGE "Hello, world!\n"

int main () {
  printf (MESSAGE);
  return 0;
}
```

Command: `gcc -S hello.c`

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world!\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-1)"
```

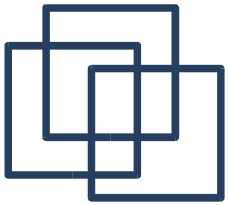


Assembly

```
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello world!\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.5 (Debian 1:3.3.5-1)"
```

Command: `as -o hello.o hello.s`

```
00000000: 7f45 4c46 0101 0100 0000 0000 0000 0000  .ELF............
00000010: 0100 0300 0100 0000 0000 0000 0000 0000  ................
00000020: dc00 0000 0000 0000 3400 0000 0000 2800  ........4.....(.
00000030: 0b00 0800 5589 e583 ec08 83e4 f0b8 0000  ....U...........
00000040: 0000 29c4 c704 2400 0000 00e8 fcff ffff  ..)...$.........
00000050: b800 0000 00c9 c300 4865 6c6c 6f20 776f  ........Hello wo
00000060: 726c 6421 0a00 0047 4343 3a20 2847 4e55  rld!...GCC: (GNU
00000070: 2920 332e 332e 3520 2844 6562 6961 6e20  ) 3.3.5 (Debian
00000080: 313a 332e 332e 352d 3129 0000 2e73 796d  1:3.3.5-1)...sym
00000090: 7461 6200 2e73 7472 7461 6200 2e73 6873  tab..strtab..shs
000000a0: 7472 7461 6200 2e72 656c 2e74 6578 7400  trtab..rel.text.
000000b0: 2e64 6174 6100 2e62 7373 002e 726f 6461  .data..bss..roda
000000c0: 7461 002e 6e6f 7465 2e47 4e55 2d73 7461  ta..note.GNU-sta
000000d0: 636b 002e 636f 6d6d 656e 7400 0000 0000  ck..comment.....
000000e0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000f0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000100: 0000 0000 1f00 0000 0100 0000 0600 0000  ................
00000110: 0000 0000 3400 0000 2300 0000 0000 0000  ....4...#.......
00000120: 0000 0000 0400 0000 0000 0000 1b00 0000  ................
00000130: 0900 0000 0000 0000 0000 0000 4c03 0000  ............L...
00000140: 1000 0000 0900 0000 0100 0000 0400 0000  ................
00000150: 0800 0000 2500 0000 0100 0000 0300 0000  ....%...........
00000160: 0000 0000 5800 0000 0000 0000 0000 0000  ....X...........
00000170: 0000 0000 0400 0000 0000 0000 2b00 0000  ............+...
00000180: 0800 0000 0300 0000 0000 0000 5800 0000  ............X...
00000190: 0000 0000 0000 0000 0000 0000 0400 0000  ................
000001a0: 0000 0000 3000 0000 0100 0000 0200 0000  ....0...........
000001b0: 0000 0000 5800 0000 0e00 0000 0000 0000  ....X...........
000001c0: 0000 0000 0100 0000 0000 0000 3800 0000  ............8...
000001d0: 0100 0000 0000 0000 0000 0000 6600 0000  ............f...
000001e0: 0000 0000 0000 0000 0000 0000 0100 0000  ................
000001f0: 0000 0000 4800 0000 0100 0000 0000 0000  ....H...........
00000200: 0000 0000 6600 0000 2500 0000 0000 0000  ....f...%.......
00000210: 0000 0000 0100 0000 0000 0000 1100 0000  ................
00000220: 0300 0000 0000 0000 0000 0000 8b00 0000  ................
00000230: 5100 0000 0000 0000 0000 0000 0100 0000  Q...............
00000240: 0000 0000 0100 0000 0200 0000 0000 0000  ................
00000250: 0000 0000 9402 0000 a000 0000 0a00 0000  ................
00000260: 0800 0000 0400 0000 1000 0000 0900 0000  ................
00000270: 0300 0000 0000 0000 0000 0000 3403 0000  ............4...
00000280: 1500 0000 0000 0000 0000 0000 0100 0000  ................
00000290: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000002a0: 0000 0000 0100 0000 0000 0000 0000 0000  ................
000002b0: 0400 f1ff 0000 0000 0000 0000 0000 0000  ................
000002c0: 0300 0100 0000 0000 0000 0000 0000 0000  ................
000002d0: 0300 0300 0000 0000 0000 0000 0000 0000  ................
000002e0: 0300 0400 0000 0000 0000 0000 0000 0000  ................
000002f0: 0300 0500 0000 0000 0000 0000 0000 0000  ................
00000300: 0300 0600 0000 0000 0000 0000 0000 0000  ................
00000310: 0300 0700 0900 0000 0000 0000 2300 0000  ............#...
00000320: 1200 0100 0e00 0000 0000 0000 0000 0000  ................
00000330: 1000 0000 0068 656c 6c6f 2e63 006d 6169  .....hello.c.mai
00000340: 6e00 7072 696e 7466 0000 0000 1300 0000  n.printf........
00000350: 0105 0000 1800 0000 0209 0000            ............
```

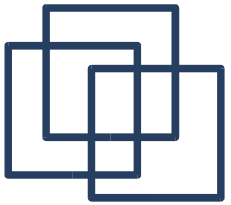


Linking

Command:

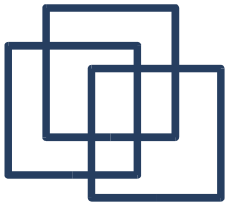
```
ld -o hello -dynamic-linker
/lib/ld-linux.so.2 /usr/lib/crt1.o
/usr/lib/crti.o
/usr/lib/gcc-lib/i686/3.3.1/crtbegin.o
-L/usr/lib/gcc-lib/i686/3.3.1 hello.o
-lgcc -lgcc_eh -lc -lgcc -lgcc_eh
/usr/lib/gcc-lib/i686/3.3.1/crtend.o
/usr/lib/crtn.o
```

Alternate Command: `gcc -o hello hello.o`



Optimization Levels

- -O0: No optimization.
- -O1: Reduce code size and execution time.
- -O2: Maximum optimization without size increasing (no loop unrolling or inlining).
- -O3: Function In-lining plus some more aggressive optimizations.
- -Os: Reduce the size of the executable as most as possible.

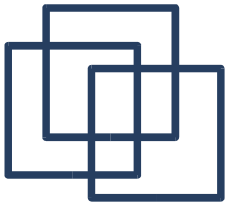


Optimization Levels

Without -O, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results.

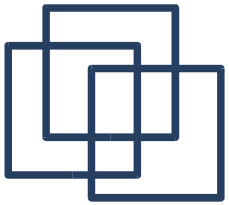
In other words, statements are independents.

If you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.



Flags Optimization Options

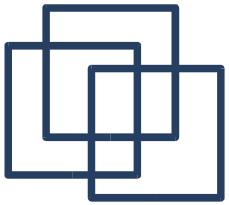
- -funroll-loops
- -fomit-frame-pointer
- -finline-funtions
- -fmerge-constants
- -fexpensive-optimizations
- -foptimize-register-move
- ... read the f**cking manual ...



-O0 Vs. -O1

```
int allocs0(int input){
  int output;

  output = 1;
  output = 2;
  output = 3;
  output = 4;

  output += input;

  return output;
}
```

```
int allocs1(int input){
  return input+4;
}
```

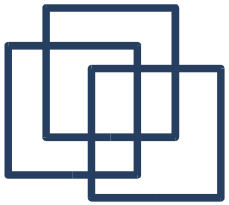


-O0 Vs. -O1

Compiled with: `-O0 -fomit-frame-pointer`

```
allocs0:
    subl   $4, %esp
    movl   $1, (%esp)
    movl   $2, (%esp)
    movl   $3, (%esp)
    movl   $4, (%esp)
    movl   8(%esp), %edx
    movl   %esp, %eax
    addl   %edx, (%eax)
    movl   (%esp), %eax
    addl   $4, %esp
    ret
```

```
allocs1:
    movl   4(%esp), %eax
    addl   $4, %eax
    ret
```

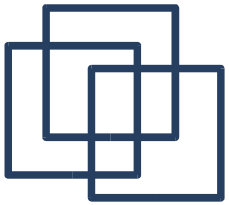


-O0 Vs. -O1

Compiled with: `-O1 -fomit-frame-pointer`

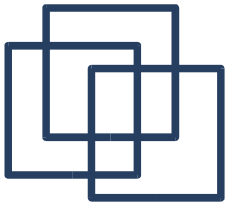
```
allocs0:
    movl   4(%esp), %eax
    addl   $4, %eax
    ret
```

```
allocs1:
    movl   4(%esp), %eax
    addl   $4, %eax
    ret
```



-O1 Vs. -O2

```c
#include <stdio.h>

int main() {
   int i, j;

   i = 5;
   j = 6;

   printf("%i %i\n", i, j);

   return 0;
}
```

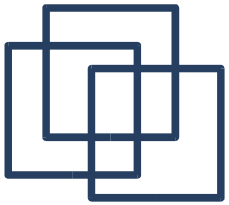


-O1 Vs. -O2

Compiled with: `-O1 -fomit-frame-pointer`

```c
#include <stdio.h>

int main() {
   int i, j;

   i = 5;
   j = 6;

   printf("%i %i\n", i, j);

   return 0;
}
```

```
.LC0:
    .string "%i %i\n"
    .text
main:
    pushl %ebp
    movl  %esp, %ebp
    subl  $24, %esp
    andl  $-16, %esp
    movl  $6, 8(%esp)
    movl  $5, 4(%esp)
    movl  $.LC0, (%esp)
    call  printf
    movl  $0, %eax
    movl  %ebp, %esp
    popl  %ebp
    ret
```

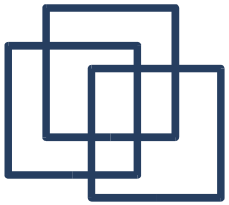


-O1 Vs. -O2

Compiled with: `-O2 -fomit-frame-pointer`

```c
#include <stdio.h>

int main() {
  int i, j;

  i = 5;
  j = 6;

  printf("%i %i\n", i, j);

  return 0;
}
```

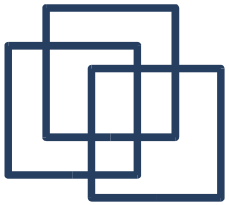
```
.LC0:
    .string "%i %i\n"
    .text
main:
    pushl   %ebp
    movl    $6, %edx
    movl    %esp, %ebp
    subl    $24, %esp
    movl    $5, %eax
    andl    $-16, %esp
    movl    %edx, 8(%esp)
    movl    %eax, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
    movl    %ebp, %esp
    xorl    %eax, %eax
    popl    %ebp
    ret
```



-O1 Vs. -O2

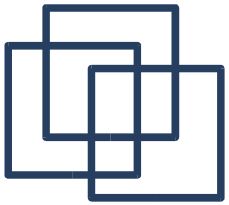
```
.LC0:
    .string "%i %i\n"
    .text
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $24, %esp
    andl   $-16, %esp
    movl   $6, 8(%esp)
    movl   $5, 4(%esp)
    movl   $.LC0, (%esp)
    call   printf
    movl   $0, %eax
    movl   %ebp, %esp
    popl   %ebp
    ret
```

```
.LC0:
    .string "%i %i\n"
    .text
main:
    pushl    %ebp
    movl     $6, %edx
    movl     %esp, %ebp
    subl     $24, %esp
    movl     $5, %eax
    andl     $-16, %esp
    movl     %edx, 8(%esp)
    movl     %eax, 4(%esp)
    movl     $.LC0, (%esp)
    call     printf
    movl     %ebp, %esp
    xorl     %eax, %eax
    popl     %ebp
    ret
```



-O2 Vs. -O3

```c
#include <stdio.h>

int foo(int n) {
  int i, result = 0;

  for (i=0; i<n; i++) {
    result = result + i;
  }

  return result;
}

int main() {

  printf("Result = %i\n", foo(3));

  return 0;
}
```

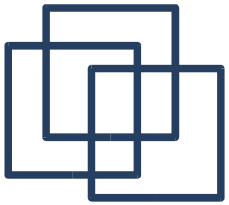


-O2 Vs. -O3

Compiled with: `-O2 -fomit-frame-pointer -Wall`

```
foo:
    movl   4(%esp), %ecx
    xorl   %eax, %eax
    xorl   %edx, %edx
    cmpl   %ecx, %eax
    jge    .L8
.L6:
    addl   %edx, %eax
    incl   %edx
    cmpl   %ecx, %edx
    jl     .L6
.L8:
    ret
```

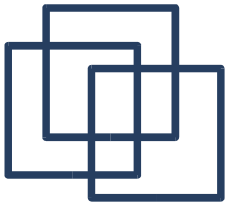
```
main:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    andl   $-16, %esp
    movl   $3, (%esp)
    call   foo
    movl   %eax, 4(%esp)
    movl   $.LC0, (%esp)
    call   printf
    movl   %ebp, %esp
    xorl   %eax, %eax
    popl   %ebp
    ret
```



-O2 Vs. -O3

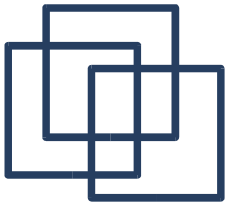
Compiled with: `-O3 -fomit-frame-pointer -Wall`

```
main:
    pushl %ebp
    xorl  %edx, %edx
    movl  %esp, %ebp
    subl  $8, %esp
    xorl  %eax, %eax
    andl  $-16, %esp
.L6:
    addl  %eax, %edx
    incl  %eax
    cmpl  $3, %eax
    jl    .L6
    movl  %edx, 4(%esp)
    movl  $.LC0, (%esp)
    call  printf
    movl  %ebp, %esp
    xorl  %eax, %eax
    popl  %ebp
    ret
```

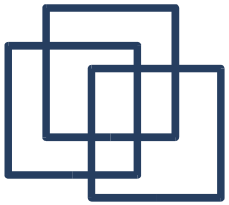


What to think about it ?

- Compilers ensure you to have a semantically equivalent code (except in case of bugs).
- You never know exactly what assembly code the compiler is producing out of your code.
- A compiler is a tool which is helpful but which shouldn't be trusted in matter of extreme optimizations.
- When it comes to extreme optimization matter, you have to look at the assembly code (whatever language or processor you are using).



Profiling with gprof

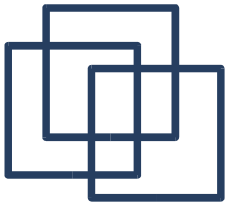


Using gprof

- Compile the software (option -pg):
  `gcc -o my_program -pg -O2 my_program.c`

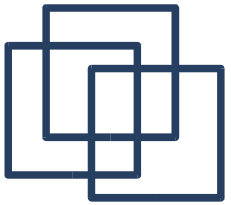
- Execute the software:
  `./my_program`

- Run gprof to analyse the file gmon.out:
  `gprof -b ./my_program`



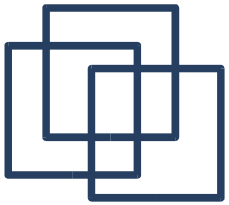
gprof Output (1)

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds  calls  ms/call  ms/call  name
 58.83      0.58     0.58                             main
 27.38      0.85     0.27        1   271.11   271.11  foo0
  7.10      0.92     0.07        1    70.29    70.29  foo1
  7.10      0.99     0.07        1    70.29    70.29  foo2
```



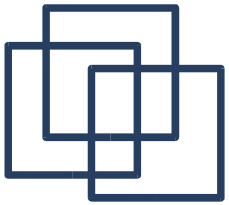
gprof Output (1)

- %time: Percentage of the total running time of the program used by this function.
- Cumulative seconds: Running sum of the total number of seconds
- Self seconds: Number of seconds accounted for by this function
- Calls: Number of times this function was invoked
- self ms/call: Average number of milliseconds spent in this function per call
- total ms/call: Average number of milliseconds spent in this function and its descendants per call



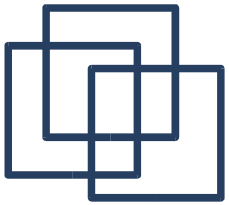
gprof Output (2)

```
index % time    self  children    called     name
                                              <spontaneous>
[1]    100.0    0.58     0.41                 main [1]
                0.28     0.00       1/1           foo0 [2]
                0.07     0.00       1/1           foo1 [3]
                0.06     0.00       1/1           foo2 [4]
-----------------------------------------------
                0.28     0.00       1/1           main [1]
[2]     28.3    0.28     0.00       1         foo0 [2]
-----------------------------------------------
                0.07     0.00       1/1           main [1]
[3]      7.1    0.07     0.00       1         foo1 [3]
-----------------------------------------------
                0.06     0.00       1/1           main [1]
[4]      6.1    0.06     0.00       1         foo2 [4]
-----------------------------------------------
```



gprof Output (2)

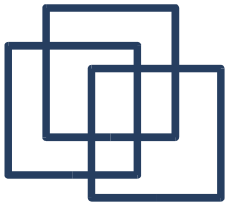
- **Index:** Unique number given to each element of the table.
- **%Time:** Percentage of the `total' time that was spent in this function and its children.
- **Self:** Total amount of time spent in this function.
- **Children:** Total amount of time propagated into this function by its children.
- **Called:** Number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.
- **Name:** Name of the current function. The index number is printed after it.



gprof Output (2)

For the function's parents, the fields have the following meanings:

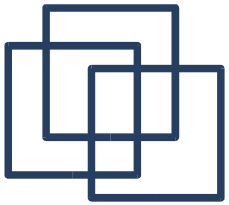
- **Self:** Amount of time propagated directly from the function into this parent.
- **Children:** Amount of time propagated from the function's children into this parent.
- **Called:** Number of times this parent called the function `/' or the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.
- **Name:** Name of the parent. If the parents of the function can't be determined, `<spontaneous>' is printed in the `name' field.



gprof Output (2)

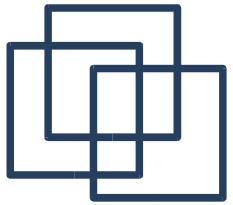
For the function's children, the fields have the following meanings:

- **Self:** Amount of time propagated directly from the child into the function.
- **Children:** Amount of time propagated from the child's children to the function.
- **Called:** Number of times the function called this child `/' the total number of times the child was called. Recursive calls by the child are not listed in the number after the `/'.
- **Name:** Name of the child. The child's index number is printed after it. If the child is a member of a cycle, the cycle number is printed between the name and the index number.



Matrix Copy

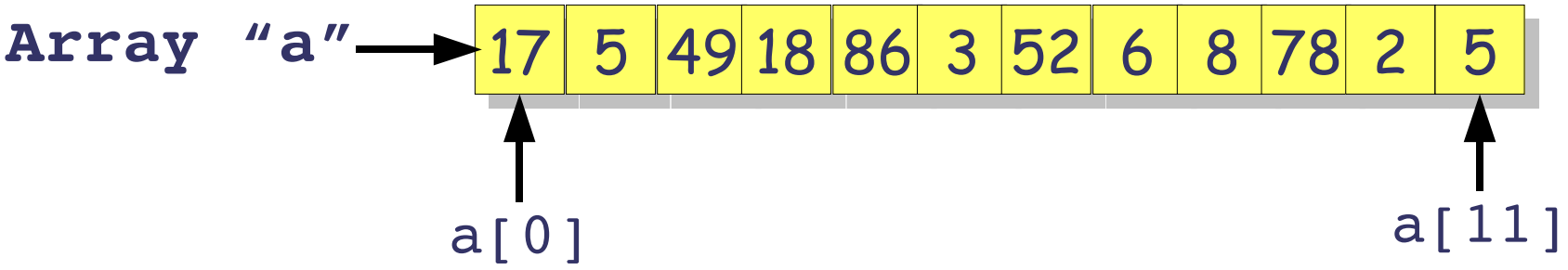
(why is locality good...)



One-dimension Array

Declaration:

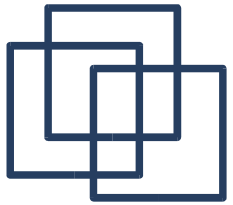
```
int a[12];
```




```
a[i] = *(a+i)
```

base address

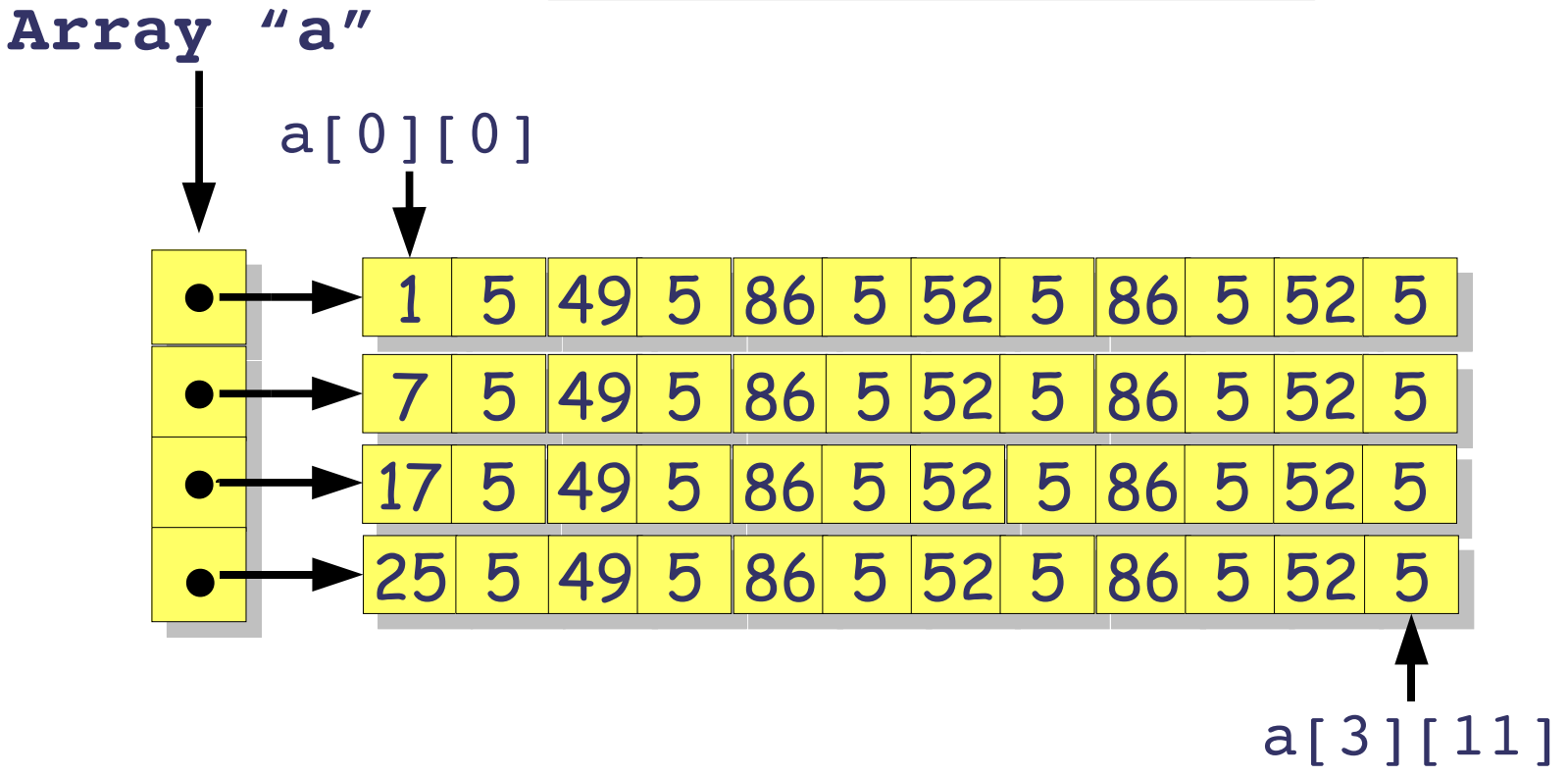
index



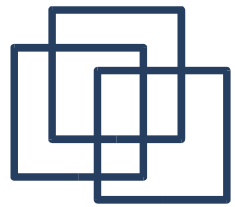
Two-dimensions Array

Declaration:

```
int a[4][12];
```

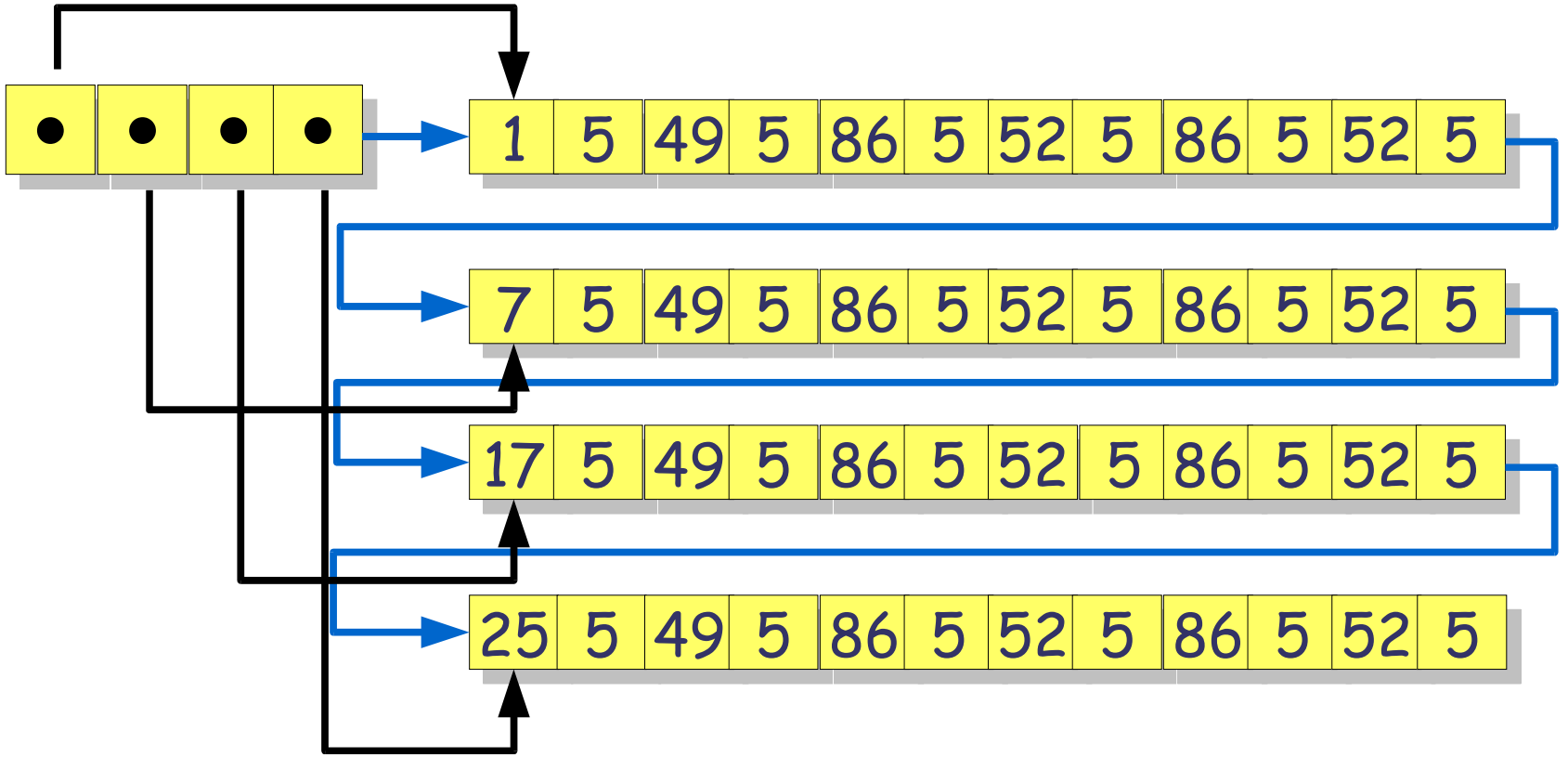


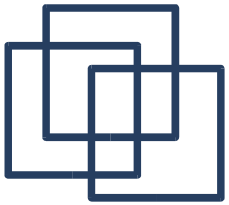

```
a[i][j] = *(a[i]+j) = *(*(a+i)+j)
```



Two-dimensions Array (2)

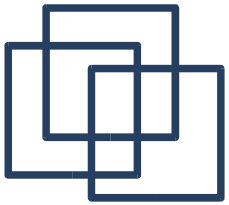
In fact, the matrix is represented as a line in the memory.





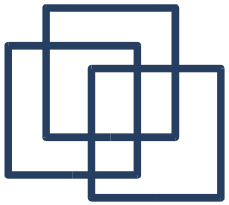
copy1

```
int copy1(float src[SIZE_X][SIZE_Y],
          float dest[SIZE_X][SIZE_Y]) {
  int i, j;

  for (j=0; j<SIZE_Y; j++)
    for (i=0; i<SIZE_X; i++)
      dest[i][j] = src[i][j];

  return 0;
}
```



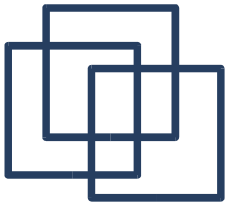
copy1

```
copy1:
   pushl %edi
   xorl  %edi, %edi
   pushl %esi
   pushl %ebx
   movl  16(%esp), %esi
   movl  20(%esp), %ebx
.L11:
   movl  %edi, %edx
   movl  $1999, %ecx
.L10:
   movl  (%esi,%edx,4), %eax
   movl  %eax, (%ebx,%edx,4)
   addl  $75, %edx
   decl  %ecx
   jns   .L10
   incl  %edi
   cmpl  $74, %edi
   jle   .L11
   popl  %ebx
   xorl  %eax, %eax
   popl  %esi
   popl  %edi
   ret
```



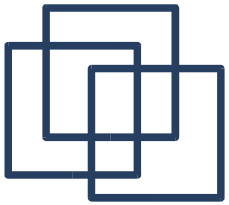
copy2

```
int copy2(float src[SIZE_X][SIZE_Y],
          float dest[SIZE_X][SIZE_Y]) {
  int i, j;

  for (i=0; i<SIZE_X; i++)
    for (j=0; j<SIZE_Y; j++)
      dest[i][j] = src[i][j];

  return 0;
}
```



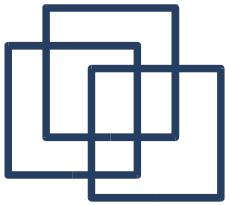
copy2

```
copy2:
        pushl   %ebp
        xorl    %ebp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        movl    20(%esp), %edi
        xorl    %ebx, %ebx
        movl    24(%esp), %esi
.L26:
        xorl    %ecx, %ecx
.L25:
        leal    (%ecx,%ebx), %edx
        movl    (%edi,%edx,4), %eax
        incl    %ecx
        cmpl    $74, %ecx
        movl    %eax, (%esi,%edx,4)
        jle     .L25
        incl    %ebp
        addl    $75, %ebx
        cmpl    $1999, %ebp
        jle     .L26
        popl    %ebx
        xorl    %eax, %eax
        popl    %esi
        popl    %edi
        popl    %ebp
        ret
```



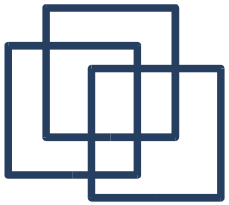
copy3

```
int copy3(float* src, float* dest) {
   int size;

   for (size=(SIZE_X*SIZE_Y); size; size--)
      *dest++ = *src++;

   return 0;
}
```



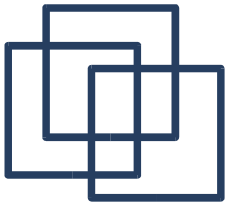
copy3

```
copy3:
        pushl   %ebx
        movl    8(%esp), %ecx
        movl    $150000, %ebx
        movl    12(%esp), %edx
        .p2align 4,,15
.L36:
        movl    (%ecx), %eax
        addl    $4, %ecx
        movl    %eax, (%edx)
        addl    $4, %edx
        decl    %ebx
        jne     .L36
        popl    %ebx
        xorl    %eax, %eax
        ret
```



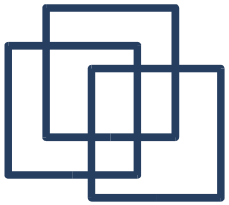
copy4

```
int copy4(float* src, float* dest) {

  memcpy(dest, src,
         (SIZE_X*SIZE_Y)*sizeof(float));

  return 0;
}
```



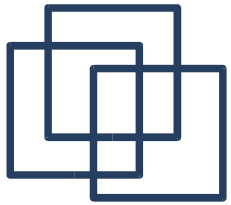
copy4

```
copy4:
        subl    $8, %esp
        movl    $600000, %eax
        movl    %edi, 4(%esp)
        movl    16(%esp), %edi
        movl    %esi, (%esp)
        movl    12(%esp), %esi
        testl   $4, %edi
        je      .L40
        movl    (%esi), %eax
        addl    $4, %esi
        movl    %eax, (%edi)
        addl    $4, %edi
        movl    $599996, %eax
.L40:
        cld
        movl    %eax, %ecx
        shrl    $2, %ecx
        rep
        movsl
        xorl    %eax, %eax
        movl    (%esp), %esi
        movl    4(%esp), %edi
        addl    $8, %esp
        ret
```

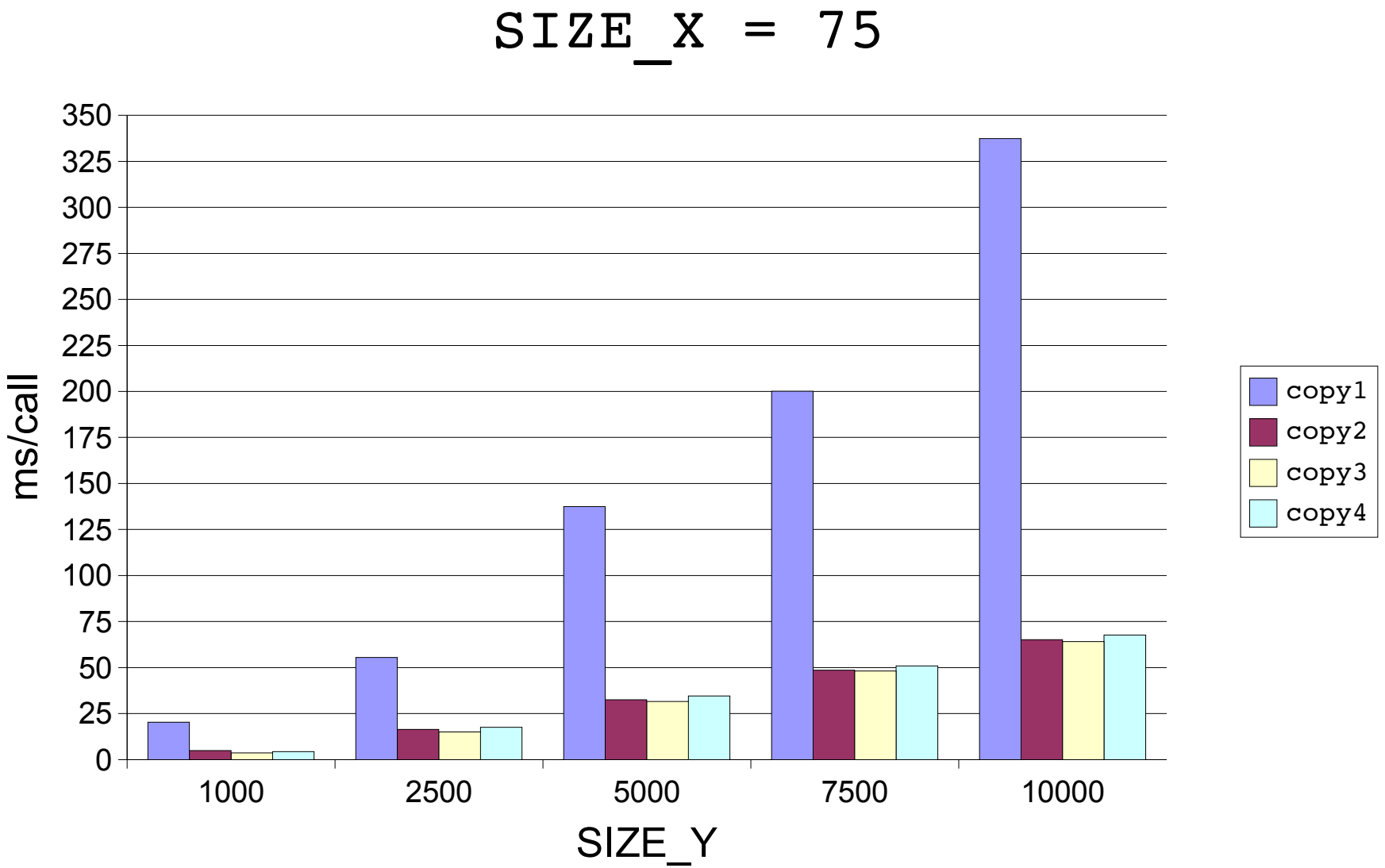


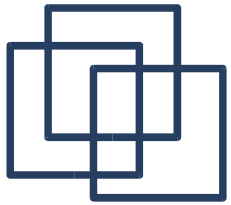
Comparing Copy Methods

```
int main() {

  int i,j,k;
  float array1[SIZE_X][SIZE_Y], array2[SIZE_X][SIZE_Y];

  for (i=0; i<SIZE_X; i++)
    for (j=0; j<SIZE_Y; j++)
      array1[i][j] =0;

  for (k=0; k<1000; k++) {
    copy1(array1, array2);
    copy3(array1[0], array2[0]);
    copy2(array1, array2);
    copy4(array1[0], array2[0]);
  }

  return 0;
}
```



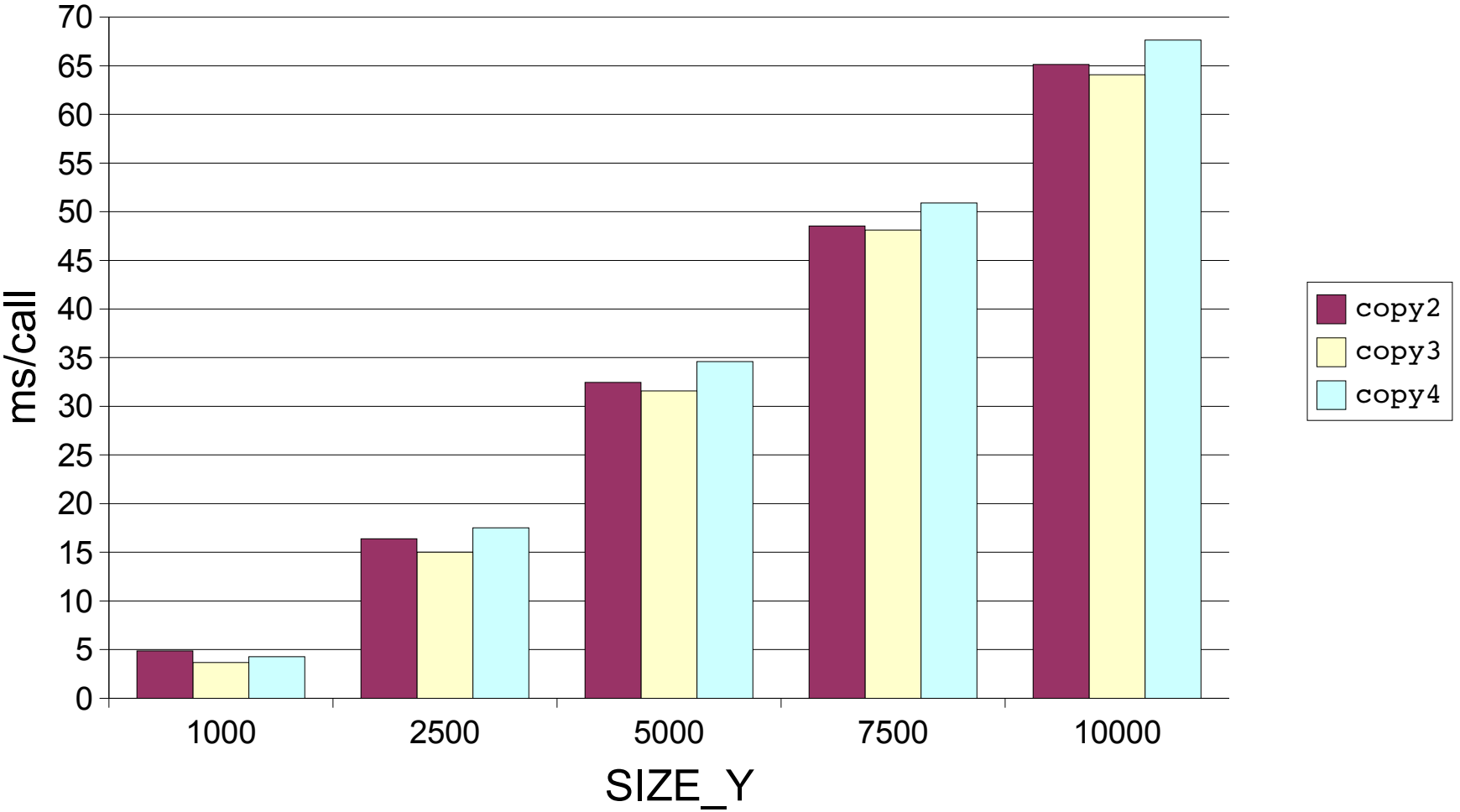
Performances (SIZE_Y)

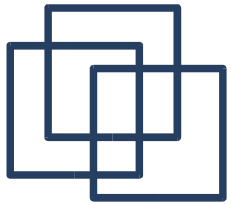




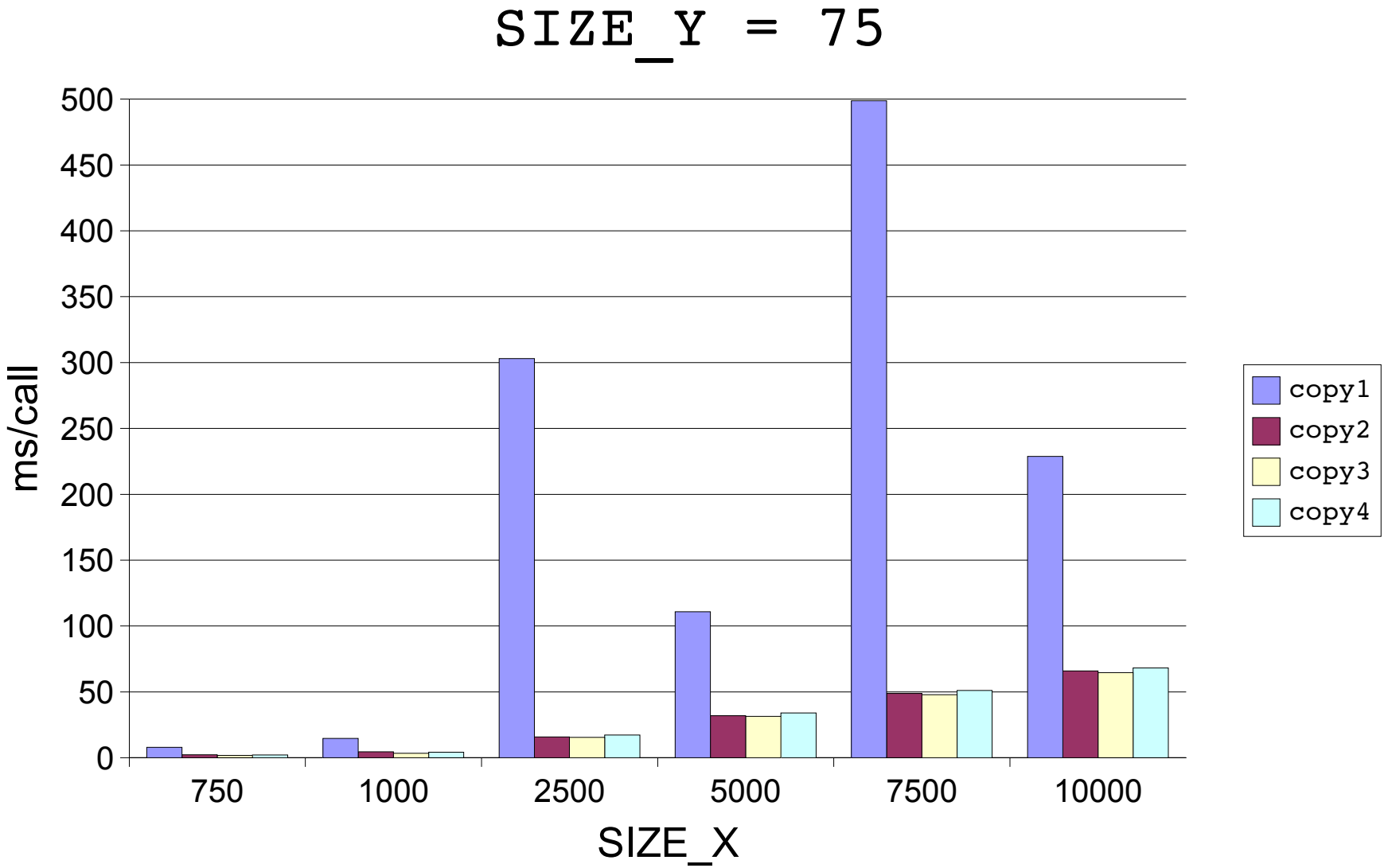
Performances (SIZE_Y)

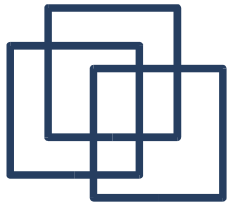
SIZE_X = 75



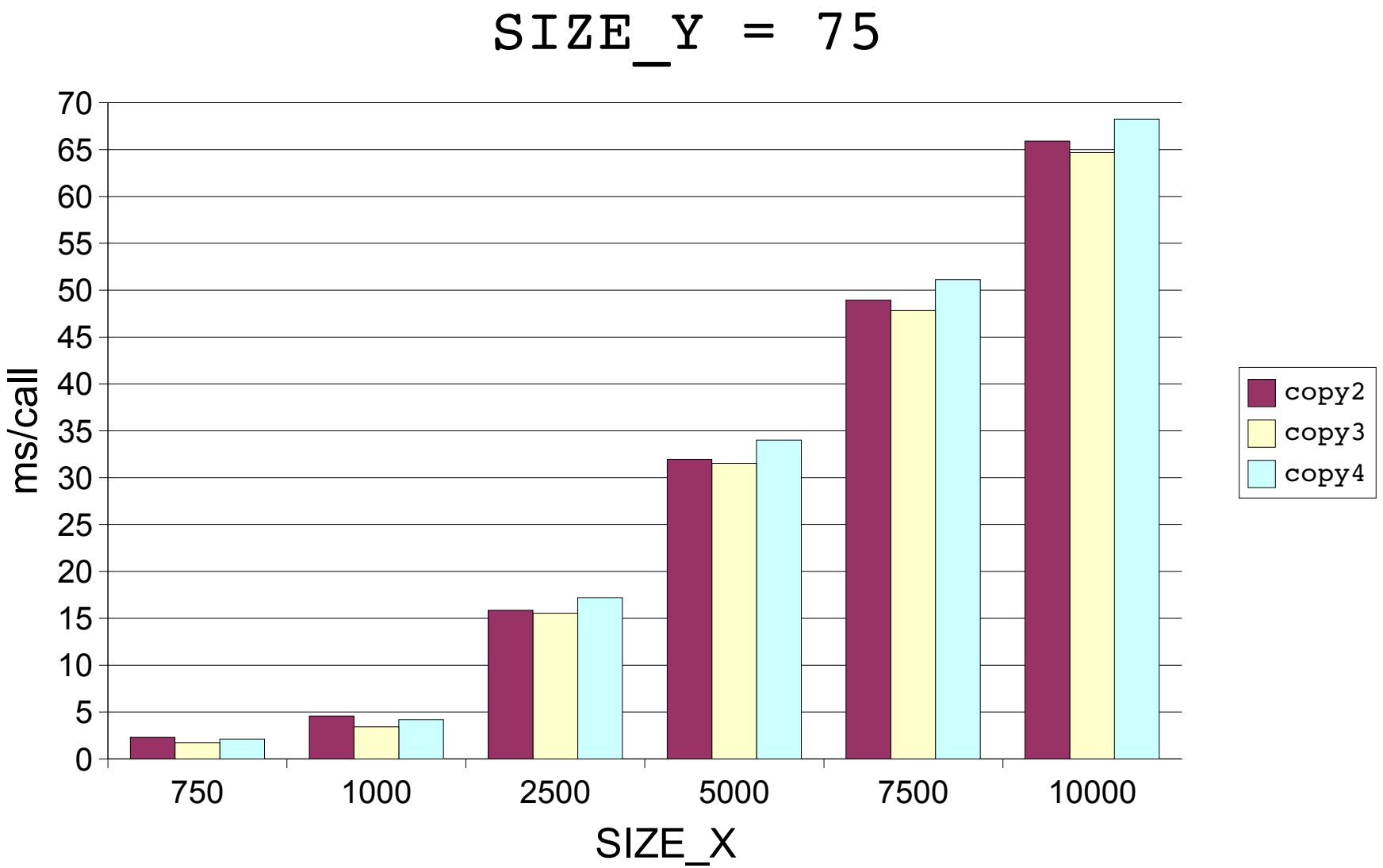


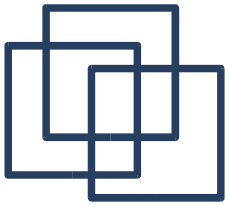
Performances (SIZE_X)





Performances (SIZE_X)



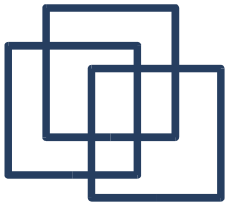


Code Optimization

If... Else if....

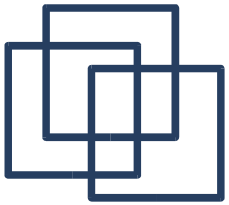
Vs

Switch



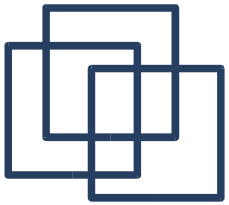
If ... Else If ...

```
int ifelseif(int input)
{
        int output = 0;

        if (input == 0) {
                output = 1;
        } else if (input == 1) {
                output = 2;
        }

         ... etc ...

        } else if (input == 15) {
                output = 16;
        }
        return output;
}
```



Switch

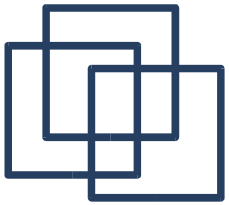
```
int switch(int input)
{
    int output = 0;

    switch (input) {
    case 0:
            output = 1;
            break;
    case 1:
            output = 2;
            break;
   ... etc ...
    case 15:
            output = 16;
            break;
    }
    return output;
}
```



if...elseif... Vs switch

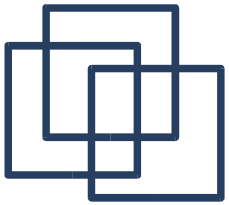
```
ifelseif:
   movl    4(%esp), %edx
   xorl    %eax, %eax
   cmpl    $0, %edx
   je      .L7
   cmpl    $1, %edx
   je      .L8
   cmpl    $2, %edx
   je      .L9
.L3:
   ret
.L9:
   movl    $3, %eax
   jmp     .L3
.L8:
   movl    $2, %eax
   ret
.L7:
   movl    $1, %eax
   ret
```

```
switch:
   movl    4(%esp), %edx
   xorl    %eax, %eax
   cmpl    $1, %edx
   je      .L14
   jg      .L17
   cmpl    $0, %edx
   je      .L12
   ret
.L12:
   movl    $1, %eax
   ret
.L17:
   cmpl    $3, %edx
   je      .L13
   ret
.L13:
   movl    $2, %eax
   ret
.L14:
   movl    $2, %eax
   ret
```



if...elseif... Vs switch

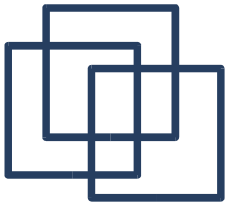
```
int main() {
  int i, output;

  srand(123456);
  for (i=0; i<ITERATIONS; i++)
    output = ifelseif((unsigned int)rand()>>24);

  srand(123456);
  for (i=0; i<ITERATIONS; i++)
    output = switch((unsigned int)rand()>>24);

  return 0;
}
```



Ifelseif Vs. Switch

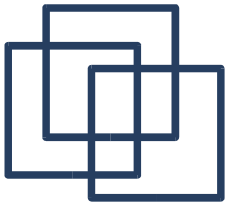
% time	self seconds	calls	self ns/call	total ns/call	name
39.78	39.92				main
25.99	26.08	100000000	260.84	260.84	ifesleif
17.96	18.03	100000000	180.27	180.27	switch

The switch is 1.5 more efficient than the if … else if…



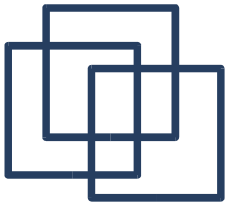
Ordered Switch

```
int switch(int input)
{
    int output = 0;

    switch (input) {
    case 15:
            output = 1;
            break;
    case 14:
            output = 2;
            break;
   ... etc ...
    case 0:
            output = 16;
            break;
    }
    return output;
}
```



switch Vs orderedswitch

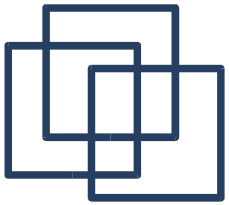
```
int main() {
  int i, output;

  srand(123456);
  for (i=0; i<ITERATIONS; i++)
    output = switch(((unsigned int)rand()+
                     (unsigned int)rand()+
                     (unsigned int)rand()+
                     (unsigned int)rand())>>26);

  srand(123456);
  for (i=0; i<ITERATIONS; i++)
    output = orderedswitch(((unsigned int)rand()+
                            (unsigned int)rand()+
                            (unsigned int)rand()+
                            (unsigned int)rand())>>26);

  return 0;
}
```



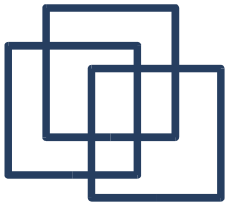
Switch Vs. Ordered Switch

% time	self seconds	calls	self ns/call	total ns/call	name
39.78	39.92				main
17.96	18.03	100000000	180.27	180.27	switch
15.86	15.91	100000000	159.12	159.12	orderedswitch

Ordering the entry by probability of occurrence does help...

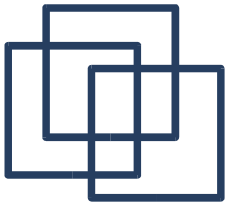


Loop Unrolling



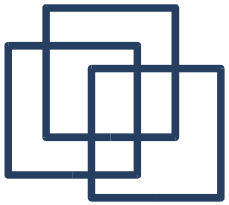
Bitvector Or (1)

```
int bv_or1(long *X, long *Y, long *Z)
{
  long bits = bits_(X);
  long size = size_(X);

  if ((size > 0) &&
      (bits == bits_(Y)) &&
      (bits == bits_(Z)))
  {
    while (size-- > 0) *X++ = *Y++ | *Z++;
    *(--X) &= mask_(X);
  }
  return 0;
}
```

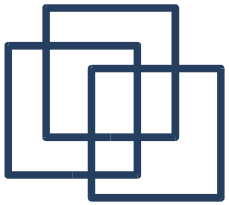


Bitvector Or (2)

```
int bv_or2(long *X, long *Y, long *Z)
{
  long bits = bits_(X);
  long size = size_(X);

  if ((size > 0) &&
      (bits == bits_(Y)) &&
      (bits == bits_(Z)))
    {
      *X = *Y | *Z;
      while (--size) *++X = *++Y | *++Z;
      *X &= mask_(X);
    }
  return 0;
}
```

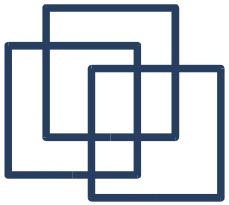


Length of an Array



Length of an Array (1)

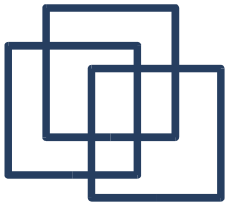
```
int length1(char **array)
{
   int i = 0;

   while ((i<SIZE) &&
          (array[i] != NULL))
      i++;

   return ((i<SIZE) ? i : -1);
}
```



Length of an Array (1)

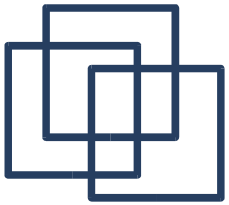
```
int length1(char **array)
{
  int i = 0;

  while ((i<SIZE) &&
         (array[i] != NULL))
    i++;

  return ((i<SIZE) ? i : -1);
}
```

```
length1:
    movl   4(%esp), %ecx
    xorl   %eax, %eax
    movl   (%ecx), %edx
    testl  %edx, %edx
    je     .L10
.L6:
    incl   %eax
    movl   $-1, %edx
    cmpl   $SIZE, %eax
    jg     .L8
    movl   (%ecx,%eax,4), %edx
    testl  %edx, %edx
    jne    .L6
.L10:
    movl   %eax, %edx
.L8:
    movl   %edx, %eax
    ret
```



Length of an Array (2)

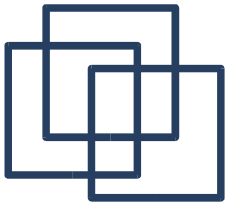
```
int length2(char **array)
{
   int i = 0;

   while ((array[i] != NULL) &&
          (i<SIZE))
      i++;

   return ((i<SIZE) ? i : -1);
}
```



Length of an Array (2)

```
int length2(char **array)
{
  int i = 0;

  while ((array[i] != NULL) &&
         (i<SIZE))
    i++;

  return ((i<SIZE) ? i : -1);
}
```

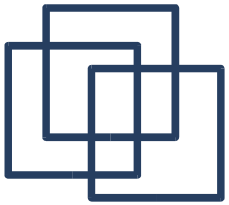
```
length2:
    movl   4(%esp), %edx
    xorl   %eax, %eax
    movl   (%edx), %ecx
    testl  %ecx, %ecx
    je     .L19
.L17:
    incl   %eax
    movl   (%edx,%eax,4), %ecx
    testl  %ecx, %ecx
    je     .L14
    cmpl   $SIZE, %eax
    jle    .L17
.L18:
    movl   $-1, %eax
.L19:
    ret
.L14:
    cmpl   $SIZE, %eax
    jle    .L19
    jmp    .L18
```



Comparing 1 & 2

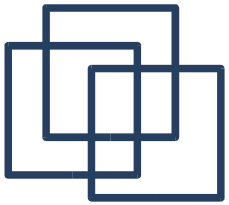
```
length1:
   movl   4(%esp), %ecx
   xorl   %eax, %eax
   movl   (%ecx), %edx
   testl  %edx, %edx
   je     .L10
.L6:
   incl   %eax
   movl   $-1, %edx
   cmpl   $SIZE, %eax
   jg     .L8
   movl   (%ecx,%eax,4), %edx
   testl  %edx, %edx
   jne    .L6
.L10:
   movl   %eax, %edx
.L8:
   movl   %edx, %eax
   ret
```

```
length2:
   movl   4(%esp), %edx
   xorl   %eax, %eax
   movl   (%edx), %ecx
   testl  %ecx, %ecx
   je     .L19
.L17:
   incl   %eax
   movl   (%edx,%eax,4), %ecx
   testl  %ecx, %ecx
   je     .L14
   cmpl   $SIZE, %eax
   jle    .L17
.L18:
   movl   $-1, %eax
.L19:
   ret
.L14:
   cmpl   $SIZE, %eax
   jle    .L19
   jmp    .L18
```



Length of an Array (3)

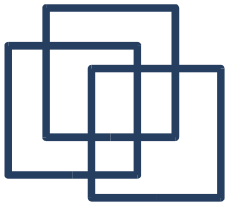
```
int length3(char **array)
{
   int i;

   for(i=0; i<SIZE; i++) {
      if (array[i] != NULL)
       continue;
      return i;
   }

   return -1;
}
```



Length of an Array (3)

```
int length3(char **array)
{
  int i;

  for(i=0; i<SIZE; i++) {
    if (array[i] != NULL)
      continue;
    return i;
  }

  return -1;
}
```

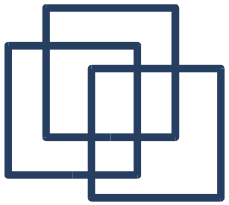
```
length3:
    movl   4(%esp), %edx
    xorl   %eax, %eax
.L29:
    movl   (%edx,%eax,4), %ecx
    testl  %ecx, %ecx
    je     .L23
    incl   %eax
    cmpl   $SIZE, %eax
    jle    .L29
    movl   $-1, %eax
.L23:
    ret
```



Comparing 1 & 3

```
length1:
   movl   4(%esp), %ecx
   xorl   %eax, %eax
   movl   (%ecx), %edx
   testl  %edx, %edx
   je     .L10
.L6:
   incl   %eax
   movl   $-1, %edx
   cmpl   $SIZE, %eax
   jg     .L8
   movl   (%ecx,%eax,4), %edx
   testl  %edx, %edx
   jne    .L6
.L10:
   movl   %eax, %edx
.L8:
   movl   %edx, %eax
   ret
```

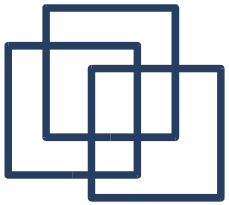
```
length3:
   movl   4(%esp), %edx
   xorl   %eax, %eax
.L29:
   movl   (%edx,%eax,4), %ecx
   testl  %ecx, %ecx
   je     .L23
   incl   %eax
   cmpl   $SIZE, %eax
   jle    .L29
   movl   $-1, %eax
.L23:
   ret
```



Comparing 2 & 3

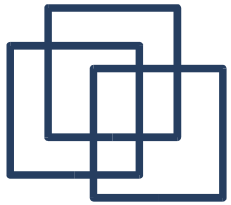
```
length2:
   movl   4(%esp), %edx
   xorl   %eax, %eax
   movl   (%edx), %ecx
   testl  %ecx, %ecx
   je     .L19
.L17:
   incl   %eax
   movl   (%edx,%eax,4), %ecx
   testl  %ecx, %ecx
   je     .L14
   cmpl   $SIZE, %eax
   jle    .L17
.L18:
   movl   $-1, %eax
.L19:
   ret
.L14:
   cmpl   $SIZE, %eax
   jle    .L19
   jmp    .L18
```

```
length3:
   movl   4(%esp), %edx
   xorl   %eax, %eax
.L29:
   movl   (%edx,%eax,4), %ecx
   testl  %ecx, %ecx
   je     .L23
   incl   %eax
   cmpl   $SIZE, %eax
   jle    .L29
   movl   $-1, %eax
.L23:
   ret
```

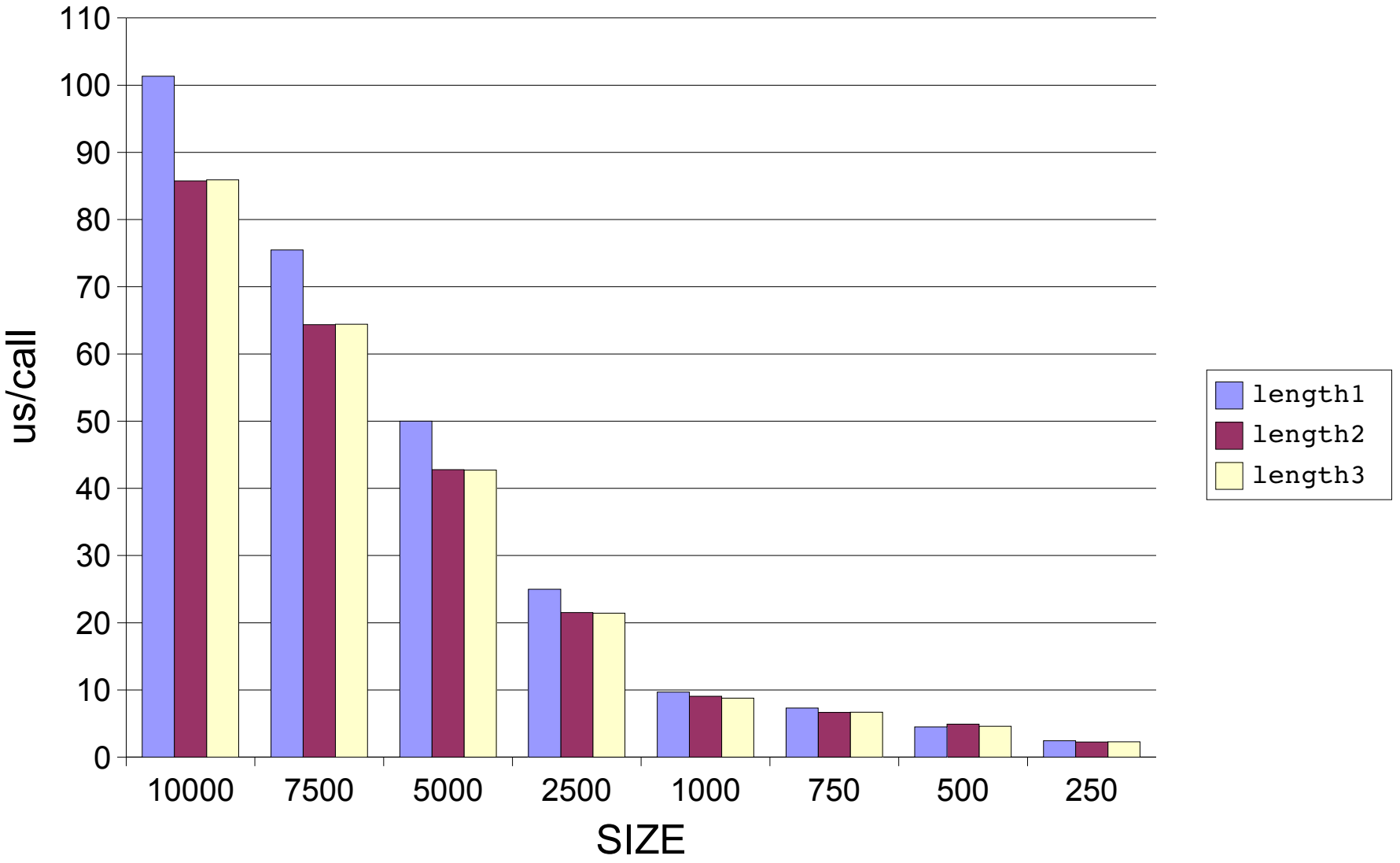


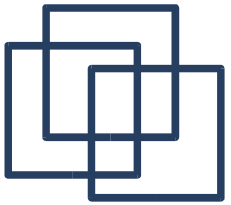
Comparing 1, 2 & 3

```
int main() {
  char* array[SIZE];
  char* string = "a";
  int i, j;

  for (i=0; i<SIZE; i++) {
    for (j=0; j<SIZE-i; j++)
      array[j] = (char*) string;

    array[SIZE-i] = NULL;

    for (j=0; j<1000; j++) {
      length1(array);
      length2(array);
      length3(array);
    }
  }

  return 0;
}
```



Performances Depending On SIZE





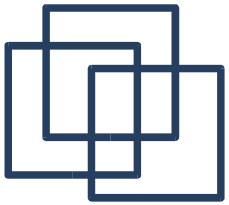
Tip: While or For Loop ?

When should we use a "for" loop better than a "while" loop and why ?

Always use a "for" loop when you already know how many iterations you will perform as you enter the loop.

Because:

- "for" loops are more readable
- Termination should be ensured

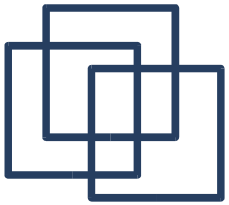


Tip: While or For Loop ?

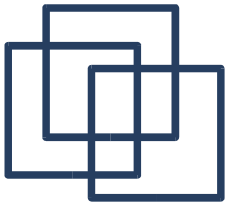
```
for (n=1000; n; n--) {
    // do your stuff
}
```

```
n = 1000;
while (n) {
    // do your stuff
    n--;
}
```

Wrong !

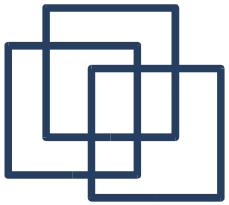


Reverse an Array



Reverse an Array (0)

```
#define SIZE 4000000

static void reverse0(char array[]) {
 int  i;
 char tmp_array[SIZE];

 for (i=0; i<SIZE; i++)
   tmp_array[SIZE-i-1] = array[i];

 for (i=0; i<SIZE; i++)
   array[i] = tmp_array[i];

 return;
}
```

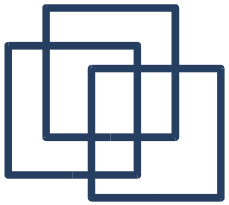


What's wrong ?

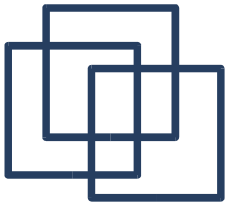
- Excessive Memory Usage:
  The function use a duplicate array.

- Excessive Number of Operations:
  Two loops are a lot of work.

- Solution: Loop Fusion !



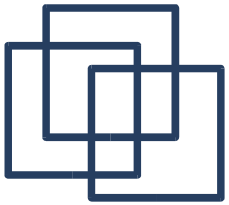
Loop Fusion



Loop Fusion

```
#define SIZE 1000000

void loop1(int a[], int b[]){
  int i;

  for (i=0; i<SIZE; i++)
    a[i] = 1;
  for (i=0; i<SIZE; i++)
    b[i] = 2;
}

void loop2(int a[], int b[]){
  int i;

  for (i=0; i<SIZE; i++) {
    a[i] = 1;
    b[i] = 2;
  }
}
```

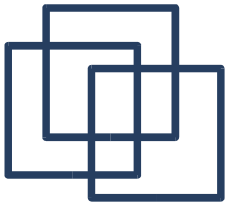
```
int main(){
  int a[SIZE], b[SIZE];

  loop1(a,b);
  loop2(a,b);

  return 0;
}
```



Loop Fusion

```
loop1:
    movl   4(%esp), %edx
    xorl   %eax, %eax
    movl   8(%esp), %ecx
.L6:
    movl   $1, (%edx,%eax,4)
    incl   %eax
    cmpl   $999999, %eax
    jle    .L6
    xorl   %eax, %eax
.L11:
    movl   $2, (%ecx,%eax,4)
    incl   %eax
    cmpl   $999999, %eax
    jle    .L11
    ret
```

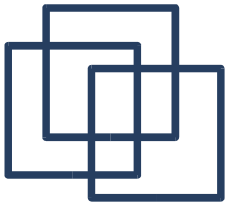
```
loop2:
    movl   4(%esp), %ecx
    xorl   %eax, %eax
    movl   8(%esp), %edx
.L21:
    movl   $1, (%ecx,%eax,4)
    movl   $2, (%edx,%eax,4)
    incl   %eax
    cmpl   $999999, %eax
    jle    .L21
    ret
```



Loop Fusion

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   self              self     total
 time seconds   calls  ms/call  ms/call  name
 61.59    0.19       1   190.94   190.94  loop1
 38.90    0.12       1   120.59   120.59  loop2
```

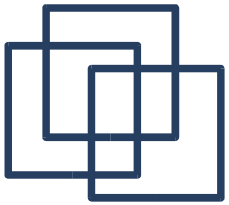
The loop fusion increase by 1.6 the efficiency of the program.



Code Optimization

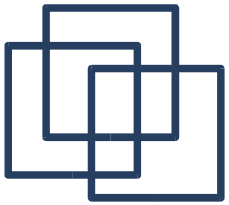
Back to

"Reverse an Array"



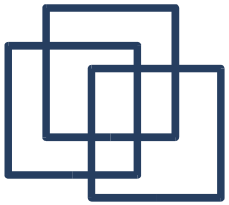
Reverse an Array (0)

```
#define SIZE 4000000

void reverse0(char array[]) {
 int  i;
 char tmp_array[SIZE];

 for (i=0; i<SIZE; i++)
   tmp_array[SIZE-i-1] = array[i];

 for (i=0; i<SIZE; i++)
   array[i] = tmp_array[i];

 return;
}
```



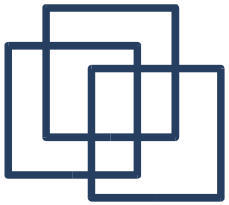
Reverse an Array (1)

```
#define SIZE 8000000

void reverse1(char array[]) {
 int  i, j;
 char tmp;

 for (i=0, j=SIZE-1; i<j; i++, j--){
   tmp = array[i];
   array[i] = array[j];
   array[j] = tmp;
 }

 return;
}
```



Comparing 0 & 1

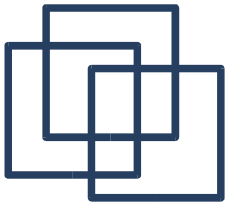
```
#define SIZE 4000000

int main() {
 int  i;
 char array[SIZE];

 for (i=0; i<SIZE; i++)
   array[i] = (char) ((int) 'a' + i%26);

 reverse0(array);
 reverse1(array);

 return 0;
}
```



gprof Output

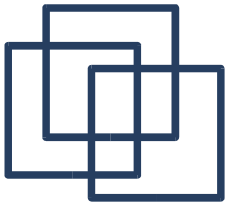
```
Flat profile:

Each sample counts as 0.01 seconds.
  %       self                 self      total
 time    seconds  calls  ms/call  ms/call  name
 58.83      0.58                             main
 27.38      0.27      1   271.11   271.11  reverse0
  7.10      0.07      1    70.29    70.29  reverse1
```

reverse1
is 3.85 more efficient
than reverse0.



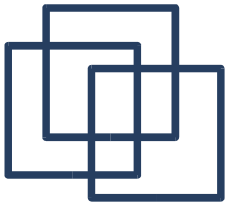
Can We Still Improve ?

```
#define SIZE 8000000

void reverse1(char array[]) {
 int  i, j;
 char tmp;

 for (i=0, j=SIZE-1; i<j; i++, j--){
   tmp = array[i];
   array[i] = array[j];
   array[j] = tmp;
 }

 return;
}
```



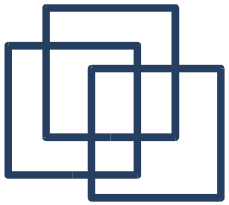
Reverse an Array (2)

```
#define SIZE 8000000

void reverse2(char array[]) {
 int  i, j;
 char tmp;

 for (i=0, j=SIZE-1; i<(SIZE>>1); i++, j--){
   tmp = array[i];
   array[i] = array[j];
   array[j] = tmp;
 }

 return;
}
```



Comparing 1 & 2

```
#define SIZE 3

int main() {
 int  i;
 char array[SIZE];

 for (i=0; i<SIZE; i++)
   array[i] = (char) ((int) 'a' + i%26);

 for (i=0; i<100000000; i++) {
 reverse1(array);
 reverse2(array);
 }

 return 0;
}
```



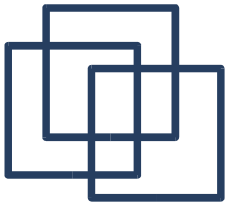
gprof Output

```
Flat profile:

Each sample counts as 0.01 seconds.
  %       self                    self     total
 time   seconds     calls  ns/call ns/call  name
 47.06    18.35 100000000  183.48  183.48  reverse1
 27.71    10.81 100000000  108.06  108.06  reverse2
```

reverse2 is

1.5 more efficient than

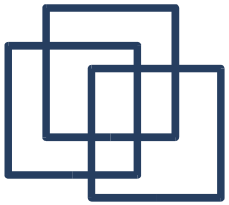
reverse1.



reverse 1 & 2 (Assembler)

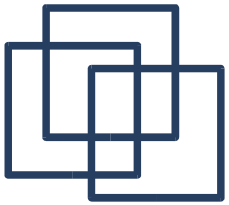
```
reverse1:
    pushl   %esi
    movl    $2, %ecx
    pushl   %ebx
    movl    12(%esp), %esi
    xorl    %ebx, %ebx
.L21:
    movzbl  (%ebx,%esi), %edx
    movzbl  (%ecx,%esi), %eax
    movb    %al, (%ebx,%esi)
    incl    %ebx
    movb    %dl, (%ecx,%esi)
    decl    %ecx
    cmpl    %ecx, %ebx
    jl      .L21
    popl    %ebx
    popl    %esi
    ret
```

```
reverse2:
    movl    4(%esp), %eax
    movzbl  (%eax), %ecx
    movzbl  2(%eax), %edx
    movb    %cl, 2(%eax)
    movb    %dl, (%eax)
    ret
```



Comparing 1 & 2 (Again)

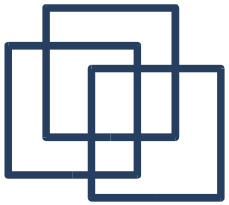
```
#define SIZE 4

int main() {
 int  i;
 char array[SIZE];

 for (i=0; i<SIZE; i++)
   array[i] = (char) ((int) 'a' + i%26);

 for (i=0; i<1000000000; i++) {
 reverse1(array);
 reverse2(array);
 }

 return 0;
}
```



Reverse an Array (2)

```
#define SIZE 8000000

void reverse2(char array[]) {
 int  i, j;
 char tmp;

 for (i=0, j=SIZE-1;
      i<(SIZE>>1);
      i++, j--){
   tmp = array[i];
   array[i] = array[j];
   array[j] = tmp;
 }

 return;
}
```

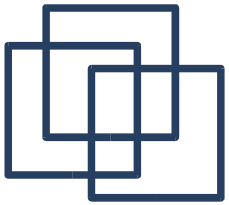
```
reverse2:
    subl    $8, %esp
    xorl    %ecx, %ecx
    movl    %esi, 4(%esp)
    movl    12(%esp), %esi
    movl    %ebx, (%esp)
    movl    $3, %ebx
.L29:
    movzbl  (%ebx,%esi), %eax
    movzbl  (%ecx,%esi), %edx
    movb    %al, (%ecx,%esi)
    incl    %ecx
    movb    %dl, (%ebx,%esi)
    decl    %ebx
    cmpl    $1, %ecx
    jle     .L29
    movl    (%esp), %ebx
    movl    4(%esp), %esi
    addl    $8, %esp
    ret
```



gprof Output

```
Flat profile:

Each sample counts as 0.01 seconds.
  %       self                    self     total
 time    seconds      calls  ns/call ns/call   name
 47.26   218.76  1000000000   218.76 218.76   reverse1
 43.68   202.19  1000000000   202.19 202.19   reverse2
```

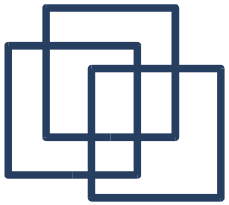
reverse2 is still
more efficient than reverse1.



reverse 1 & 2 (Assembler)

```
reverse1:
    pushl   %esi
    movl    $2, %ecx
    pushl   %ebx
    movl    12(%esp), %esi
    xorl    %ebx, %ebx
.L21:
    movzbl  (%ebx,%esi), %edx
    movzbl  (%ecx,%esi), %eax
    movb    %al, (%ebx,%esi)
    incl    %ebx
    movb    %dl, (%ecx,%esi)
    decl    %ecx
    cmpl    %ecx, %ebx
    jl      .L21
    popl    %ebx
    popl    %esi
    ret
```

```
reverse2:
    subl    $8, %esp
    xorl    %ecx, %ecx
    movl    %esi, 4(%esp)
    movl    12(%esp), %esi
    movl    %ebx, (%esp)
    movl    $3, %ebx
.L29:
    movzbl  (%ebx,%esi), %eax
    movzbl  (%ecx,%esi), %edx
    movb    %al, (%ecx,%esi)
    incl    %ecx
    movb    %dl, (%ebx,%esi)
    decl    %ebx
    cmpl    $1, %ecx
    jle     .L29
    movl    (%esp), %ebx
    movl    4(%esp), %esi
    addl    $8, %esp
    ret
```

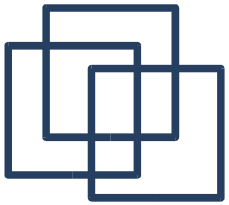


reverse 1 & 2 (Assembler)

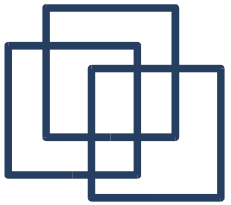
```
reverse1:
    pushl   %esi
    movl    $2, %ecx
    pushl   %ebx
    movl    12(%esp), %esi
    xorl    %ebx, %ebx
.L21:
    movzbl  (%ebx,%esi), %edx
    movzbl  (%ecx,%esi), %eax
    movb    %al, (%ebx,%esi)
    incl    %ebx
    movb    %dl, (%ecx,%esi)
    decl    %ecx
    cmpl    %ecx, %ebx
    jl      .L21
    popl    %ebx
    popl    %esi
    ret
```

```
reverse2:
    subl    $8, %esp
    xorl    %ecx, %ecx
    movl    %esi, 4(%esp)
    movl    12(%esp), %esi
    movl    %ebx, (%esp)
    movl    $3, %ebx
.L29:
    movzbl  (%ebx,%esi), %eax
    movzbl  (%ecx,%esi), %edx
    movb    %al, (%ecx,%esi)
    incl    %ecx
    movb    %dl, (%ebx,%esi)
    decl    %ebx
    cmpl    $1, %ecx
    jle     .L29
    movl    (%esp), %ebx
    movl    4(%esp), %esi
    addl    $8, %esp
    ret
```

Branch Prediction



Append to a Linked List

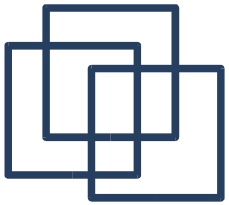


Append to a Linked List

Problem:

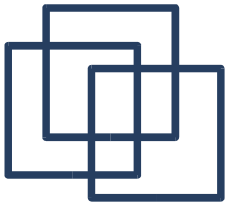
Append an element at the end of a linked list.

```
struct list {
   int index;
   struct list *next;
};
```



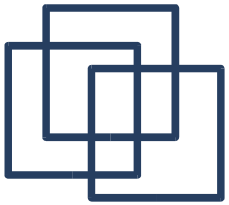
append0

```
int append0(struct list *head, struct list *new){
  if (head != NULL) {
    while (head != NULL) {
      if (head->next != NULL)
        head = head->next;
      else {
        head->next = new;
        break;
      }
    }
  } else {
    head = new;
  }
  return 0;
}
```



append1

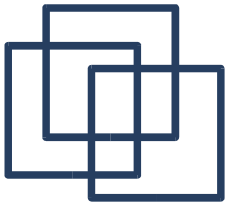
```
int append1(struct list *head, struct list *new){
  struct list *tmp = NULL;

  if (head != NULL) {
    do {
      tmp = head;
      head = head->next;
    } while (head != NULL);
    tmp->next = new;

  } else {
    head = new;
  }
  return 0;
}
```



append0 Vs. append1

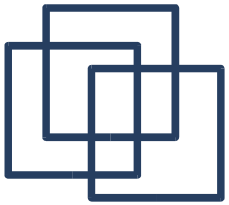
```
append0:
    movl    4(%esp), %edx
    testl   %edx, %edx
    je      .L9
.L8:
    movl    4(%edx), %eax
    testl   %eax, %eax
    je      .L12
    movl    %eax, %edx
    jmp     .L8
.L12:
    movl    8(%esp), %eax
    movl    %eax, 4(%edx)
.L9:
    xorl    %eax, %eax
    ret
```

```
append1:
    movl    4(%esp), %eax
    testl   %eax, %eax
    je      .L19
.L15:
    movl    %eax, %edx
    movl    4(%eax), %eax
    testl   %eax, %eax
    jne     .L15
    movl    8(%esp), %eax
    movl    %eax, 4(%edx)
.L19:
    xorl    %eax, %eax
    ret
```



Testing Performances

```
int main() {
  struct list *head = NULL;
  struct list elt1, elt2, elt3, elt4;
  int i;

  elt1.index = 1;
  elt2.index = 2;
  elt3.index = 3;
  elt4.index = 4;
  elt4.next = NULL;

  for (i=0; i<LOOPS; i++) {
    head = NULL;
    elt1.next = NULL;
    elt2.next = NULL;
    elt3.next = NULL;
    append0(head, &elt1);
    append0(head, &elt2);
    append0(head, &elt3);
    append0(head, &elt4); }

  for (i=0; i<LOOPS; i++) {
    head = NULL;
    elt1.next = NULL;
    elt2.next = NULL;
    elt3.next = NULL;
    append1(head, &elt1);
    append1(head, &elt2);
    append1(head, &elt3);
    append1(head, &elt4); }

  return 0; }
```



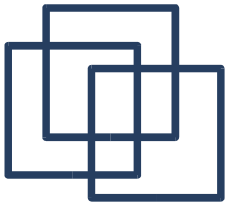
gprof Output

% time	self seconds	calls	self ns/call	total ns/call	name
31.62	11.79	400000000	29.48	29.48	append0
26.65	9.94	400000000	24.85	24.85	append1

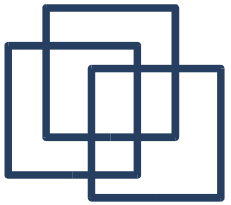
append1 is

1.2 times more efficient

than append0



Hash Functions Performances

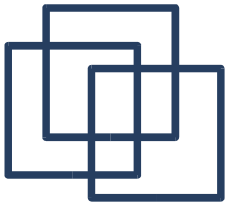


Performance of Hashing ?

Important points to check are:

- Efficiency to compute one hash:
  How much instructions/CPU cycles does it takes to compute one hash ?
- Number of Collisions:
  Giving a typical sample of data, how many collisions are observed ?

The final result should be a trade-off between this two parameters.

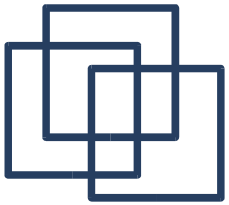


Case Study

- Problem:

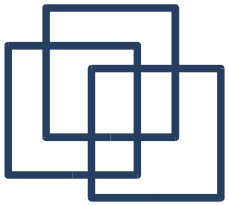
  Hashing paths in a file system

- Two hashing functions:
  - hash1
  - hash2
- Which one to choose ?



hash1

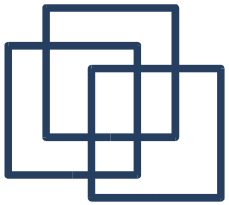
```
unsigned long hash1 (char *string) {
  unsigned long hash = 0;
  int len = strlen(string);
  int i = 0;

  for (i = 0; i < len; i++) {
    if (i % 2)
      hash = hash * ((int) string[i]);
    else
      hash = hash + ((int) string[i]);
  }

  return hash;
}
```



hash1 (Assembler)

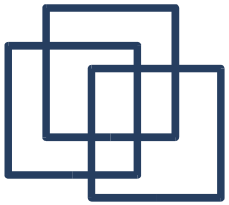
```
unsigned long hash1 (char *string){
  unsigned long hash = 0;
  int i = 0;

  for (i=0; i<strlen(string); i++){
    if (i % 2)
      hash=hash*((int) string[i]);
    else
      hash=hash+((int) string[i]);
  }

  return hash;
}
```

```
hash1:
    pushl   %esi
    pushl   %ebx
    subl    $4, %esp
    xorl    %ebx, %ebx
    movl    16(%esp), %esi
    movl    %esi, (%esp)
    call    strlen
    xorl    %edx, %edx
    movl    %eax, %ecx
    cmpl    %eax, %ebx
    jge     .L10
.L8:
    testb   $1, %dl
    je      .L6
    movsbl  (%edx,%esi),%eax
    imull    %eax, %ebx
.L4:
    incl    %edx
    cmpl    %ecx, %edx
    jl      .L8
.L10:
    popl    %edx
    movl    %ebx, %eax
    popl    %ebx
    popl    %esi
    ret
.L6:
    movsbl  (%edx,%esi),%eax
    addl    %eax, %ebx
    jmp     .L4
```



hash2

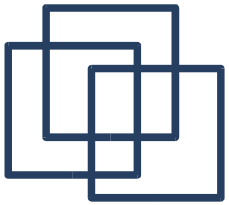
```
unsigned long hash2(char *string) {
  unsigned long hash=0;
  int len;

  for (len=strlen(string); len>0; len--) {
    hash =
      (hash + ((*string)<<4)
            + ((*string++)>>4))*11;
  }

  return (unsigned long) hash;
}
```



hash2 (Assembler)

```
unsigned long hash2(char *string) {
  unsigned long hash=0;
  int len;

  for (len=strlen(string); len>0; len--) {
    hash =
      (hash + ((*string)<<4)
            + ((*string++)>>4))*11;
  }

  return (unsigned long) hash;
}
```

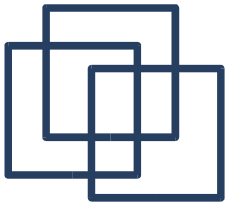
```
hash2:
   pushl  %esi
   xorl   %esi, %esi
   pushl  %ebx
   subl   $4, %esp
   movl   16(%esp), %ebx
   movl   %ebx, (%esp)
   call   strlen
   testl  %eax, %eax
   movl   %eax, %ecx
   jle    .L18
.L16:
   movzbl (%ebx), %edx
   decl   %ecx
   incl   %ebx
   movsbl %dl,%eax
   sall   $4, %eax
   sarb   $4, %dl
   leal   (%eax,%esi), %eax
   movsbl %dl,%edx
   addl   %edx, %eax
   leal   (%eax,%eax,4), %edx
   testl  %ecx, %ecx
   leal   (%eax,%edx,2), %esi
   jg     .L16
.L18:
   popl   %ecx
   movl   %esi, %eax
   popl   %ebx
   popl   %esi
   ret
```



hash1 Vs. hash2

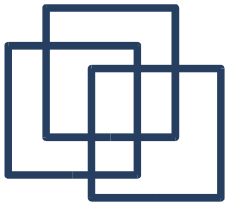
```
hash1:
   pushl   %esi
   pushl   %ebx
   subl    $4, %esp
   xorl    %ebx, %ebx
   movl    16(%esp), %esi
   movl    %esi, (%esp)
   call    strlen
   xorl    %edx, %edx
   movl    %eax, %ecx
   cmpl    %eax, %ebx
   jge     .L10
.L8:
   testb   $1, %dl
   je      .L6
   movsbl  (%edx,%esi),%eax
   imull    %eax, %ebx
.L4:
   incl    %edx
   cmpl    %ecx, %edx
   jl      .L8
.L10:
   popl    %edx
   movl    %ebx, %eax
   popl    %ebx
   popl    %esi
   ret
.L6:
   movsbl  (%edx,%esi),%eax
   addl    %eax, %ebx
   jmp     .L4
```

```
hash2:
   pushl   %esi
   xorl    %esi, %esi
   pushl   %ebx
   subl    $4, %esp
   movl    16(%esp), %ebx
   movl    %ebx, (%esp)
   call    strlen
   testl   %eax, %eax
   movl    %eax, %ecx
   jle     .L18
.L16:
   movzbl  (%ebx), %edx
   decl    %ecx
   incl    %ebx
   movsbl  %dl,%eax
   sall    $4, %eax
   sarb    $4, %dl
   leal    (%eax,%esi), %eax
   movsbl  %dl,%edx
   addl    %edx, %eax
   leal    (%eax,%eax,4), %edx
   testl   %ecx, %ecx
   leal    (%eax,%edx,2), %esi
   jg      .L16
.L18:
   popl    %ecx
   movl    %esi, %eax
   popl    %ebx
   popl    %esi
   ret
```



hash3

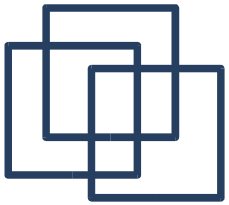
```
unsigned long hash3(char *string) {
  unsigned long hash=0;
  int len;

  for (len=strlen(string); len; len--) {
    hash =
      (hash + ((*string)<<4)
            + ((*string++)>>4))*11;
  }

  return (unsigned long) hash;
}
```



hash2 Vs. hash3

```
hash2:
   pushl   %esi
   xorl    %esi, %esi
   pushl   %ebx
   subl    $4, %esp
   movl    16(%esp), %ebx
   movl    %ebx, (%esp)
   call    strlen
   testl   %eax, %eax
   movl    %eax, %ecx
   jle     .L18
.L16:
   movzbl  (%ebx), %edx
   decl    %ecx
   incl    %ebx
   movsbl  %dl,%eax
   sall    $4, %eax
   sarb    $4, %dl
   leal    (%eax,%esi), %eax
   movsbl  %dl,%edx
   addl    %edx, %eax
   leal    (%eax,%eax,4), %edx
   testl   %ecx, %ecx
   leal    (%eax,%edx,2), %esi
   jg      .L16
.L18:
   popl    %ecx
   movl    %esi, %eax
   popl    %ebx
   popl    %esi
   ret
```

```
hash3:
   pushl   %esi
   xorl    %esi, %esi
   pushl   %ebx
   subl    $4, %esp
   movl    16(%esp), %ebx
   movl    %ebx, (%esp)
   call    strlen
   testl   %eax, %eax
   movl    %eax, %ecx
   je      .L26
.L24:
   movzbl  (%ebx), %edx
   incl    %ebx
   movsbl  %dl,%eax
   sall    $4, %eax
   sarb    $4, %dl
   leal    (%eax,%esi), %eax
   movsbl  %dl,%edx
   addl    %edx, %eax
   leal    (%eax,%eax,4), %edx
   decl    %ecx
   leal    (%eax,%edx,2), %esi
   jne     .L24
.L26:
   popl    %ebx
   movl    %esi, %eax
   popl    %ebx
   popl    %esi
   ret
```

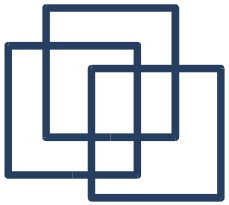


hash2 Vs. hash3

```
hash2:
    pushl   %esi
    xorl    %esi, %esi
    pushl   %ebx
    subl    $4, %esp
    movl    16(%esp), %ebx
    movl    %ebx, (%esp)
    call    strlen
    testl   %eax, %eax
    movl    %eax, %ecx
    jle     .L18
.L16:
    movzbl  (%ebx), %edx
    decl    %ecx
    incl    %ebx
    movsbl  %dl,%eax
    sall    $4, %eax
    sarb    $4, %dl
    leal    (%eax,%esi), %eax
    movsbl  %dl,%edx
    addl    %edx, %eax
    leal    (%eax,%eax,4), %edx
    testl   %ecx, %ecx
    leal    (%eax,%edx,2), %esi
    jg      .L16
.L18:
    popl    %ecx
    movl    %esi, %eax
    popl    %ebx
    popl    %esi
    ret
```

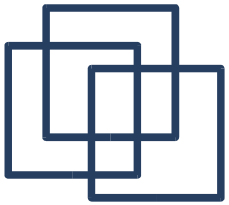
Removed

```
hash3:
    pushl   %esi
    xorl    %esi, %esi
    pushl   %ebx
    subl    $4, %esp
    movl    16(%esp), %ebx
    movl    %ebx, (%esp)
    call    strlen
    testl   %eax, %eax
    movl    %eax, %ecx
    je      .L26
.L24:
    movzbl  (%ebx), %edx
    incl    %ebx
    movsbl  %dl,%eax
    sall    $4, %eax
    sarb    $4, %dl
    leal    (%eax,%esi), %eax
    movsbl  %dl,%edx
    addl    %edx, %eax
    leal    (%eax,%eax,4), %edx
    decl    %ecx
    leal    (%eax,%edx,2), %esi
    jne     .L24
.L26:
    popl    %ebx
    movl    %esi, %eax
    popl    %ebx
    popl    %esi
    ret
```



Profiling hashing

```
int main() {
  char string[LINE_MAX];
  FILE *fp_input, *fp_output;

  if (((fp_input = fopen("./list.txt", "r")) == NULL) ||
      ((fp_output = fopen("./hash.txt", "w")) == NULL)){
    printf("File list.txt not found or cannot create hash.txt");
    return 1;
  } else {
    while ((fgets(string, LINE_MAX, fp_input)) != NULL) {
      fprintf(fp_output, "hash1: %i\n", hash1(string));
      fprintf(fp_output, "hash2: %i\n", hash2(string));
      fprintf(fp_output, "hash3: %i\n", hash3(string));
     }
    fclose(fp_input);
    fclose(fp_output);
  }
  return 0;
}
```



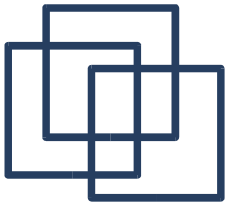
gprof Output

Hashing my home directory paths:

% time	self seconds	calls	self us/call	total us/call	name
38.07	0.22	47468	4.65	4.65	hash1
36.33	0.21	47468	4.44	4.44	hash2
25.95	0.15	47468	3.17	3.17	hash3

Hashing the whole file system paths:

% time	self seconds	calls	self us/call	total us/call	name
39.76	0.73	214587	3.39	3.39	hash2
34.55	0.63	214587	2.95	2.95	hash3
24.40	0.45	214587	2.08	2.08	hash1



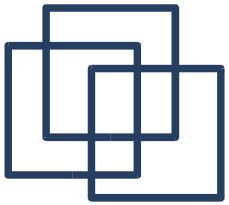
Number Of Collisions

Hashing my home directory paths:

- `hash1`: 158 Collisions
- `hash2` & `hash3`: 1 Collision

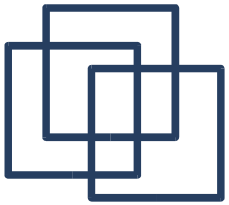
Hashing the whole file system paths:

- `hash1`: 644 Collisions
- `hash2` & `hash3`: 150 Collisions



Conclusion

- When do code optimization ?
- What is a "safe" optimization ?
- What method should I use ?



When do code optimization ?

"Premature optimization is the root of all evil."

Toni Hoare

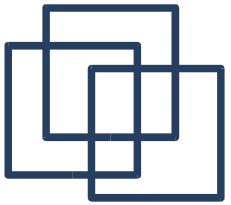
"Rules for optimization:

- Rule 1: Don't do it.
- Rule 2 (for experts only): Don't do it yet."

M.A. Jackson

Start to think about optimization when your code is working.

NOT BEFORE !!!

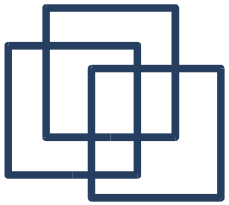


What is a "safe" optimization ?

> "Optimization always bust things, because all optimizations are, in the long haul, a form of cheating, and cheaters eventually get caught."
>
> Larry Wall (Perl's creator)

A code optimization is "safe" when:

- It doesn't affect the logic of the function
- It is done in an high-level language (no assembly in-lining)
- It doesn't obfuscate the code



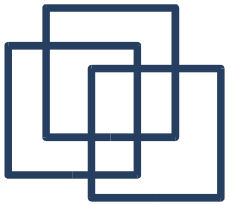
Theory is nice but ...

Don't speculate – Benchmark !

Dan Bernstein

- Computers are extremely complex
- Architectures of processors very different
- When you think that some modifications might improve the program....

CHECK IT !



What method should I use ?

1. Be logic when you code

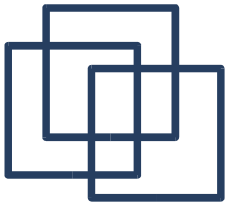
(avoid redundancy of tests, variables, actions, avoid too high complexity when not needed)

2. Don't trust the compiler

(read the assembly code generated by your compiler)

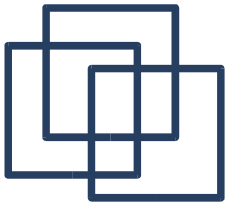
3. Test & Profile your code

(theorizing on performances has some limits, check reality from time to time)



Code Optimization

Questions ?



TODO

- Using gcov (big part missing here)
- Compare access to array vs linked-list
- Split in three lectures (parts) on:
  - "Profiling Tools"
    gprof, gcov, oprofile
  - "Memory Optimization"
    Influence of data-structure on programs,
    impact of malloc and free, data locality and alignment of data
  - "Code Optimization"
    Influence of code on programs