DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CONCURRENT AND REAL TIME PROGRAMMING
[INQ0091623] AA 2021-22

# Lab 4

# Compilers and Optimization

Gabriele Manduchi <gabriele.manduchi@unipd.it>

Andrea Rigoni Garola <andrea.rigonigarola@unipd.it>

# How the compiler works

# How compilers work

Compilers are utility programs that take your code and transform it into executable machine instruction files.

Phases of the compilation process:

1. When you run a compiler on your code, first, the **preprocessor** reads the source code (the C++ file you just wrote). The preprocessor searches for any preprocessor directives (#). Preprocessor change your code (macros, includes, hints for the compiler).

2. Next, the compiler pass through the preprocessed code line by line translating each line into the appropriate machine language instruction.
    a. Find syntax errors
    b. Perform Code Optimization

3. Finally, if no errors are present, the compiler creates an **object file** with the machine language binary necessary to run on your machine.

GCC EXAMPLE:

```
$> cd src/lab4
$> gcc ackermann.c -c
```

# Link

In order for you to have a final executable program, another utility known as the **linker** must combine your object files with the library functions necessary to run the code.

within the GNU C toolchain the linker is known as ***ld*** however all the compilation is automatically performed by a “gcc” call.

GCC EXAMPLE:

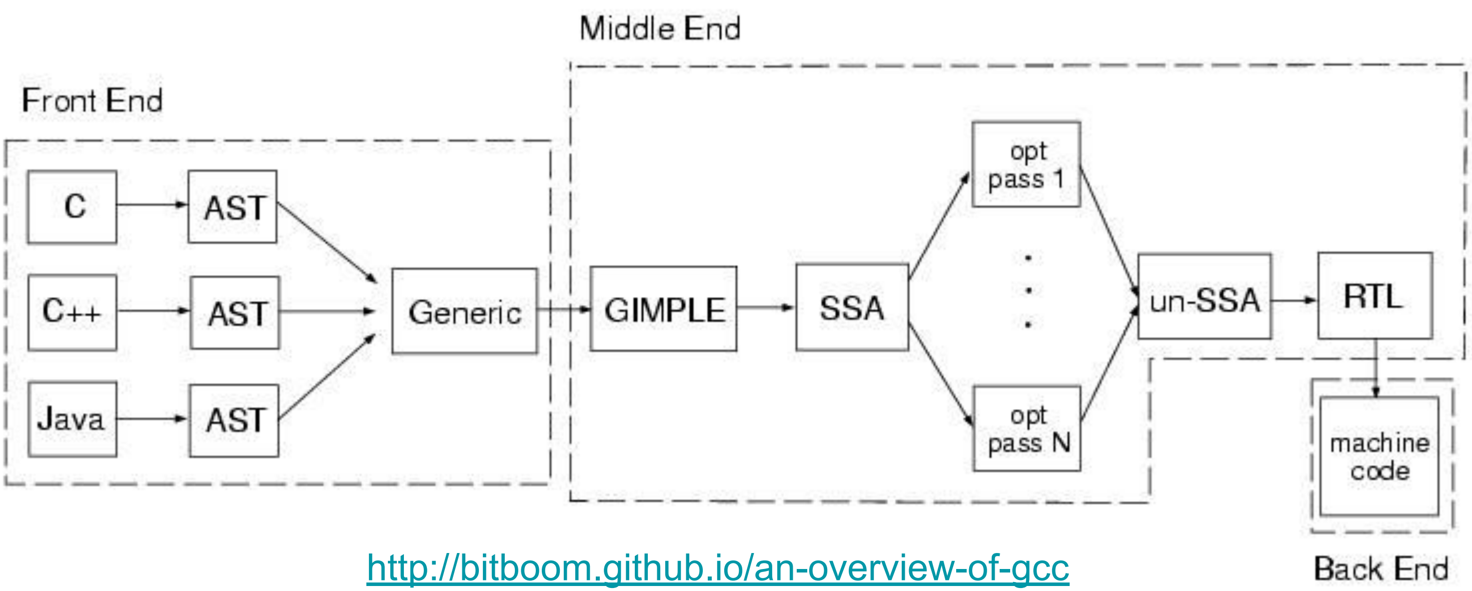
```
$> gcc -v ackernann.o -o ackermann
```

-v added to see the actual linker command

# Gnu C Compiler

The purpose of the front end is to read the source file, parse it, and convert it into the standard abstract syntax tree (AST) representation. There is one front end for each programming language

Then the compiler uses three main intermediate languages to represent the program during compilation: GENERIC, GIMPLE and RTL




http://bitboom.github.io/an-overview-of-gcc

https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html

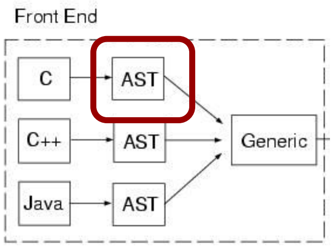
# Optimization passes

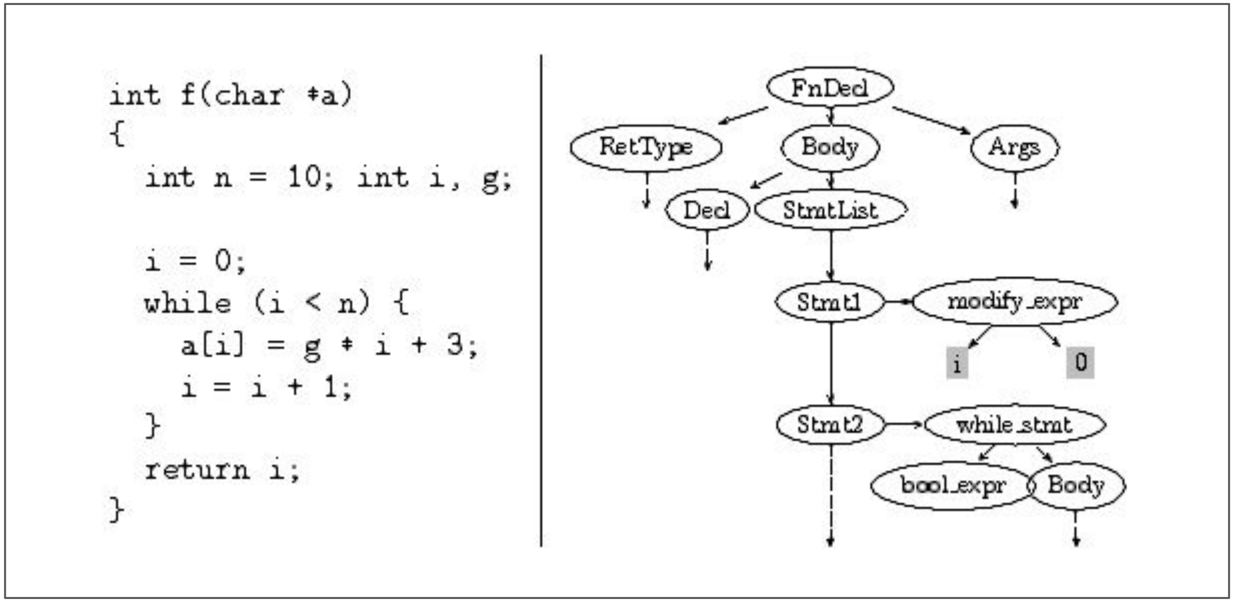
This is the very general overview of the optimization and code generation passes of the compiler.

1. **Parsing pass:** The language front end turns text into bits.
2. **Gimplification pass:** The bits are turned into something we can optimize.
3. Pass manager: Sequencing the optimization passes.
4. **IPA passes**: Inter-procedural optimizations.
5. **Tree SSA passes**: Optimizations on a high-level representation.
6. **RTL passes**: Optimizations on a low-level representation.
7. Optimization info: Dumping optimization information from passes

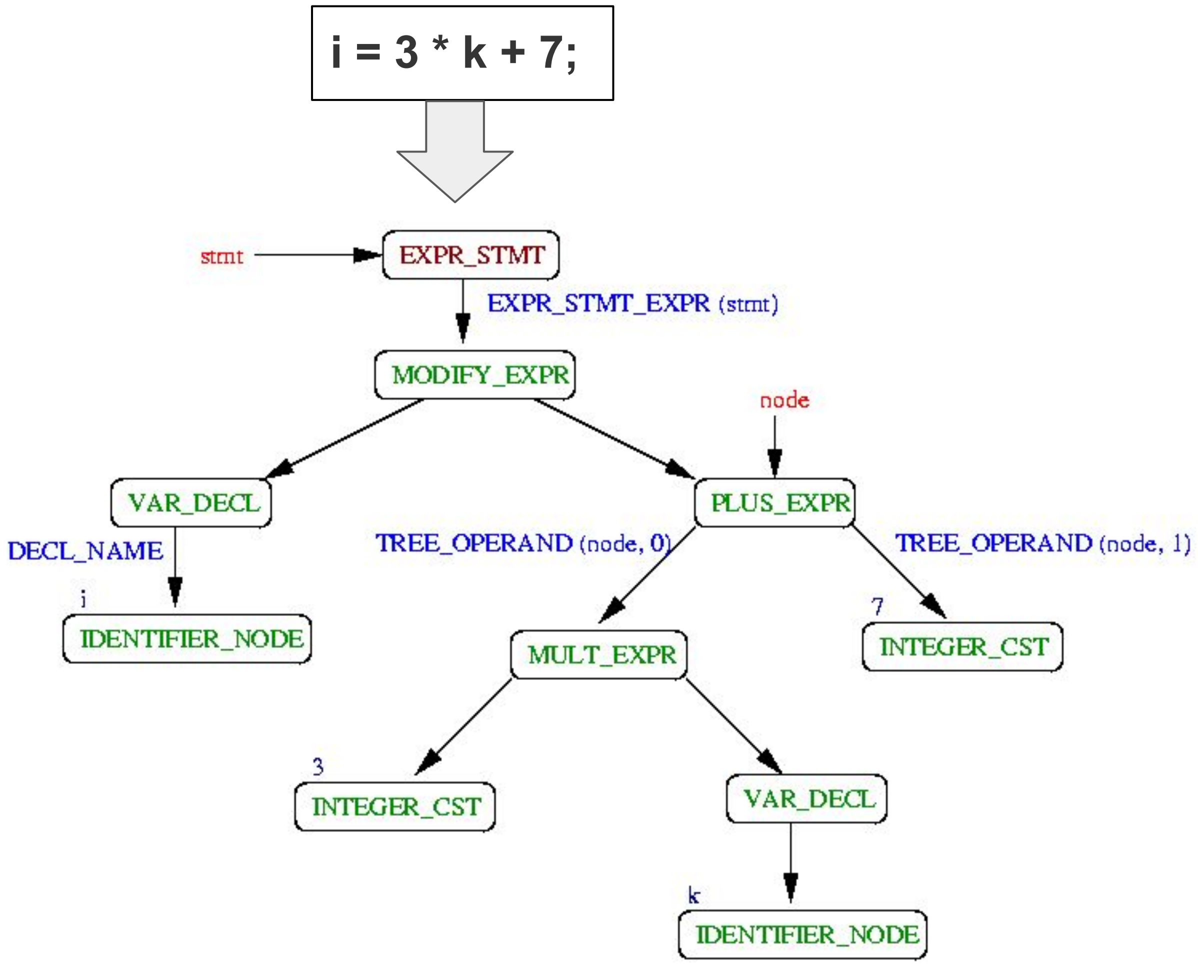
# Abstract Syntax Tree (AST)

- Tree representation of the abstract syntactic structure
- All the information provided by the programmer code
- Everything concerning the **control flow**
- Everything about **structures and types**

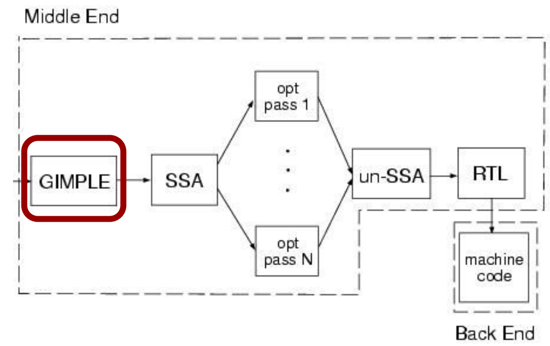


# Example: AST of a function



# Intermediate representations: GIMPLE




GIMPLE is a three-address representation derived from GENERIC by breaking down GENERIC expressions into tuples of no more than 3 operands (with some exceptions like function calls)

- Intermediate Representation, IR
- A convenient representations for optimizing the source code
- A subset of the AST/Generic
- Use only the sequencing and branching control flow constructs
- No more than three operands
- Control flow representation
    - Conditional statements
    - goto operators
    - Function calls

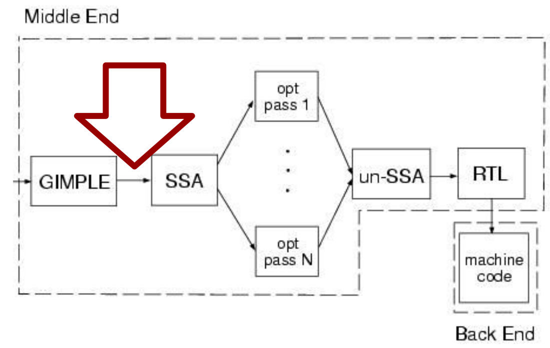
**C**

```
while (a<=7) {
    a = a + 1;
}
```

**GIMPLE**

```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
    goto <D.1196>;
else
    goto <D.1198>;
<D.1198>:;
```

# GIMPLE CFG: Control Flow Graph




The **control flow graph** is a data structure built on top of a GCC IR (the RTL or GIMPLE instruction stream) abstracting the control flow behavior of a function that is being compiled.

The CFG is a directed graph where the vertices represent basic blocks and edges represent possible transfer of control flow from one basic block to another.

**Basic Block:** A basic block is a straight-line sequence of code with only one entry point and only one exit.

## GIMPLE

```
goto <D.1197>;
<D.1196>:;
a = a + 1;
<D.1197>:;
if (a <= 7)
    goto <D.1196>;
else
    goto <D.1198>;
<D.1198>:;
```

## GIMPLE CFG bb

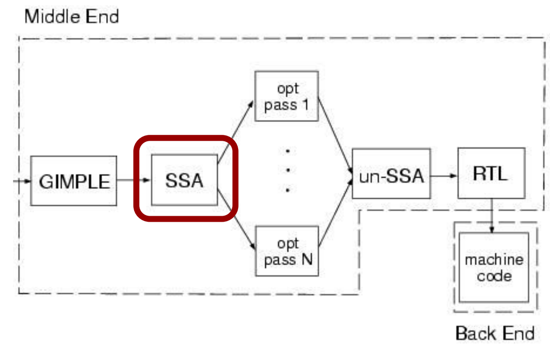
```
;; CFG annotations
 <bb 1> :
    goto <D.1197>;
    <D.1196>:;
    a = a + 1;
    <D.1197>:;
    if (a <= 7)
        goto <D.1196>;
    else
        goto <D.1198>;
    <D.1198>:;
```

# Intermediate representations: SSA

Most of the tree optimizers rely on the data flow information provided by the **Static Single Assignment** (SSA) form.

The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable.

The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created.




**GIMPLE bb**

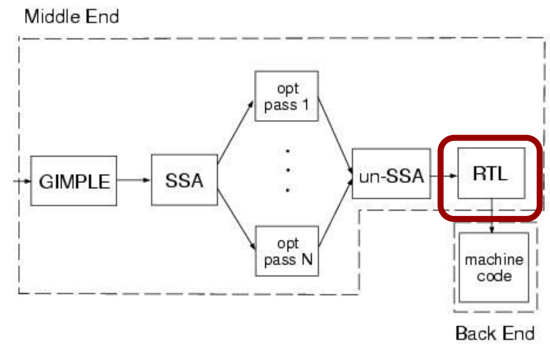
```
int f(int n) {
<bb 3> :
    int a;
    if (n>0)
        a = 1;
    else if(n<0)
        a = -1;
    else
        a = 0;
    return a;
}
```

**SSA**

```
int f(int n) {
<bb 3> :
    int a;
    if (n>0)
        a_1 = 1;
    else if(n<0)
        a_2 = -1;
    else
        a_3 = 0;
   # a_4 = PHI <a_1, a_2, a_3>
    return a_4;
}
```

# Intermediate representations: RTL

The last part of the compiler work is done on a low-level intermediate representation called **Register Transfer Level** (RTL). In this language, the instructions to be output are described in an algebraic form that describes what the instruction does.




**RTL EXAMPLE**

```
(set (reg:SI 140)
     (plus:SI (reg:SI 138)
              (reg:SI 139)))
```

*“sum the contents of register 138 with the contents of register 139 and store the result in register 140”*

# Show passes in an example program

# Example code: the ackermann function

CODING EXAMPLE

CODING EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

# define LOG_FATAL_ERROR (1)
# define LOG_ERROR       (2)
# define LOG_WARNING     (3)
# define LOG_DEBUG       (4)
# define LOG_INFO        (5)

typedef unsigned char error_level_t;
static error_level_t log_level = 0;

static void print_log(const char *format, error_level_t err_level,
                      va_list args ) {

   if (err_level < log_level) return;
   else switch (err_level)
   {
   case LOG_FATAL_ERROR:
       printf("[FATAL ERROR] ");
       break;
   case LOG_ERROR:
       printf("[ERROR] ");
       break;
```

# Example code: the ackermann function

CODING EXAMPLE

CODING EXAMPLE

```
    case LOG_WARNING:
        printf("[WARNING] ");
        break;
    case LOG_DEBUG:
        printf("[DEBUG] ");
        break;
    case LOG_INFO:
    default:
        printf("[INFO] ");
        break;    }
    vprintf (format, args);
}

#define __DEFINE_PRINT_LOG(name, errno) \
static inline void name(const char *msg, ...) { \
    va_list args; \
    va_start (args, msg); \
    print_log(msg, errno, args); \
    va_end (args); \
}
__DEFINE_PRINT_LOG(log_fatal, LOG_FATAL_ERROR)
__DEFINE_PRINT_LOG(log_error, LOG_ERROR)
__DEFINE_PRINT_LOG(log_warning, LOG_WARNING)
__DEFINE_PRINT_LOG(log_debug, LOG_DEBUG)
__DEFINE_PRINT_LOG(log_info, LOG_INFO)
#undef __DEFINE_PRINT_LOG
```

# Example code: the ackermann function

```
// https://en.wikipedia.org/wiki/Ackermann_function
static int ackermann(int m, int n){
 if (m == 0) return n + 1;
 else if (n == 0) return ackermann(m - 1, 1);
 else return ackermann(m - 1, ackermann(m, n - 1));
}

int main(int argc, char **argv){
  if( argc<2) {
     log_error("usage: %s n\n",argv[0]);
     exit(1);  }
 int n = atoi(argv[1]);
 int count = 0, total = 0, multiplied = 0;

 while(count < n){
   count += 1;
   multiplied *= count;
   if (multiplied < 100) log_info("count: %d\n",count);
   total += ackermann(2, 2);
   total += ackermann(multiplied, n);
   int d1 = ackermann(n, 1);
   total += d1 * multiplied;
   int d2 = ackermann(n, count);
   if (count % 2 == 0) total += d2;
 }
 return total;
}
```

CODING EXAMPLE

CODING EXAMPLE

# NOTE: The Ackermann function

It is one of the simplest and earliest-discovered examples of a total computable function that is **not primitive recursive**.

All primitive recursive functions are total and computable, but the Ackermann function illustrates that not all total computable functions are primitive recursive.

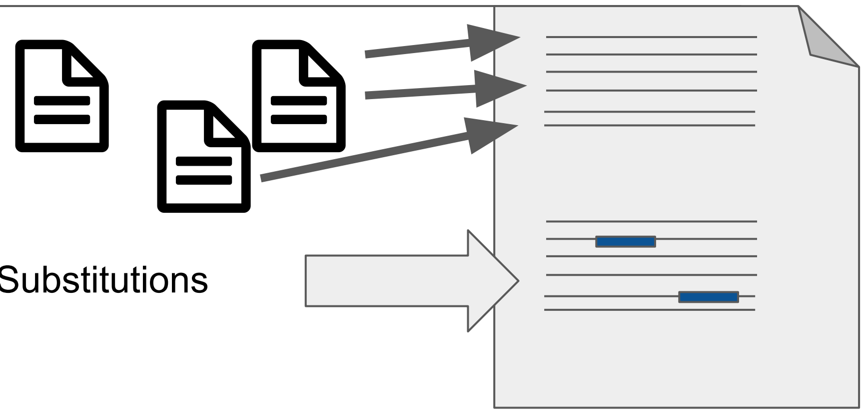
$$A(m,n) = \begin{cases} n+1 & \text{if } m=0 \\ A(m-1,1) & \text{if } m>0 \text{ and } n=0 \\ A(m-1,A(m,n-1)) & \text{if } m>0 \text{ and } n>0 \end{cases}$$

where m and n are non-negative integers

# Preprocessing

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
```

```
# define LOG_FATAL_ERROR ( 1)
# define LOG_ERROR       ( 2)
# define LOG_WARNING     ( 3)
# define LOG_DEBUG       ( 4)
# define LOG_INFO        ( 5)
```




- The preprocessor include the code from external files ( usually called headers ).

**NOTE:** The .h extension is not mandatory

- Then it proceed with macro substitutions

```
#define __DEFINE_PRINT_LOG(name, errno) \
static inline void name(const char *msg, ...) { \
   va_list args; \
   va_start (args, msg); \
   print_log(msg, errno, args); \
   va_end (args); \
}
__DEFINE_PRINT_LOG(log_fatal, LOG_FATAL_ERROR)
__DEFINE_PRINT_LOG(log_error, LOG_ERROR)
__DEFINE_PRINT_LOG(log_warning, LOG_WARNING)
__DEFINE_PRINT_LOG(log_debug, LOG_DEBUG)
__DEFINE_PRINT_LOG(log_info, LOG_INFO)
#undef __DEFINE_PRINT_LOG
```

MACRO DEFINITION

SUBSTITUTIONS

# Ackermann GIMPL:

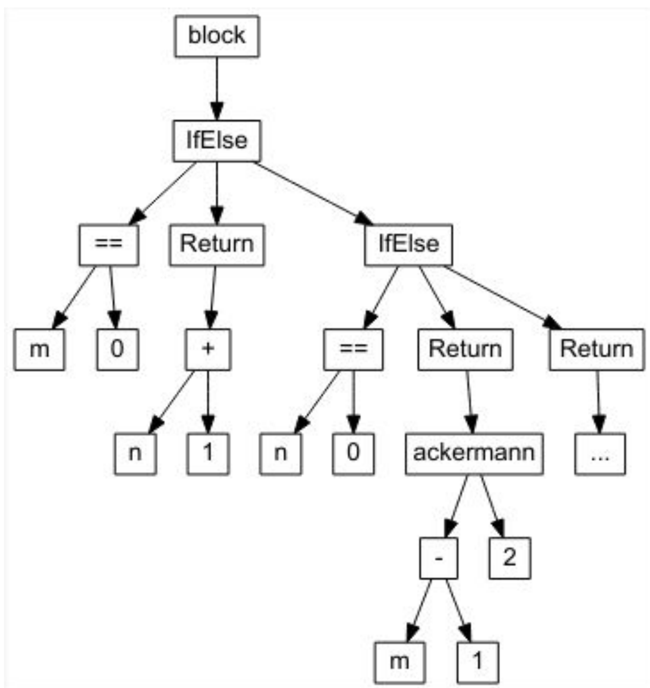
### CODE ( ackermann_function.c )

```c
int main(void) {
   return ackermann(2,2);
}

static int ackermann(int m, int n){
 if (m == 0) return n + 1;
 else if (n == 0) return ackermann(m - 1, 1);
 else return ackermann(m - 1, ackermann(m, n - 1));
}
```

### GIMPL

```c
int main ()  {
 int D.1950;
 {
   D.1950 = ackermann (2, 2);
   return D.1950;
 }
 D.1950 = 0;
 return D.1950;
}

int ackermann (int m, int n) {
 int D.1954;

 if (m == 0) goto <D.1952>; else goto <D.1953>;
 <D.1952>:
 D.1954 = n + 1;
 // predicted unlikely by early return (on trees) predictor.
 return D.1954;
 <D.1953>:
 if (n == 0) goto <D.1955>; else goto <D.1956>;
 <D.1955>:
 _1 = m + -1;
 D.1954 = ackermann (_1, 1);
 // predicted unlikely by early return (on trees) predictor.
 return D.1954;
 <D.1956>:
 _2 = n + -1;
 _3 = ackermann (m, _2);
 _4 = m + -1;
 D.1954 = ackermann (_4, _3);
 // predicted unlikely by early return (on trees) predictor.
 return D.1954;
}
```

### AST



# Ackermann CFG: Basic blocks definition

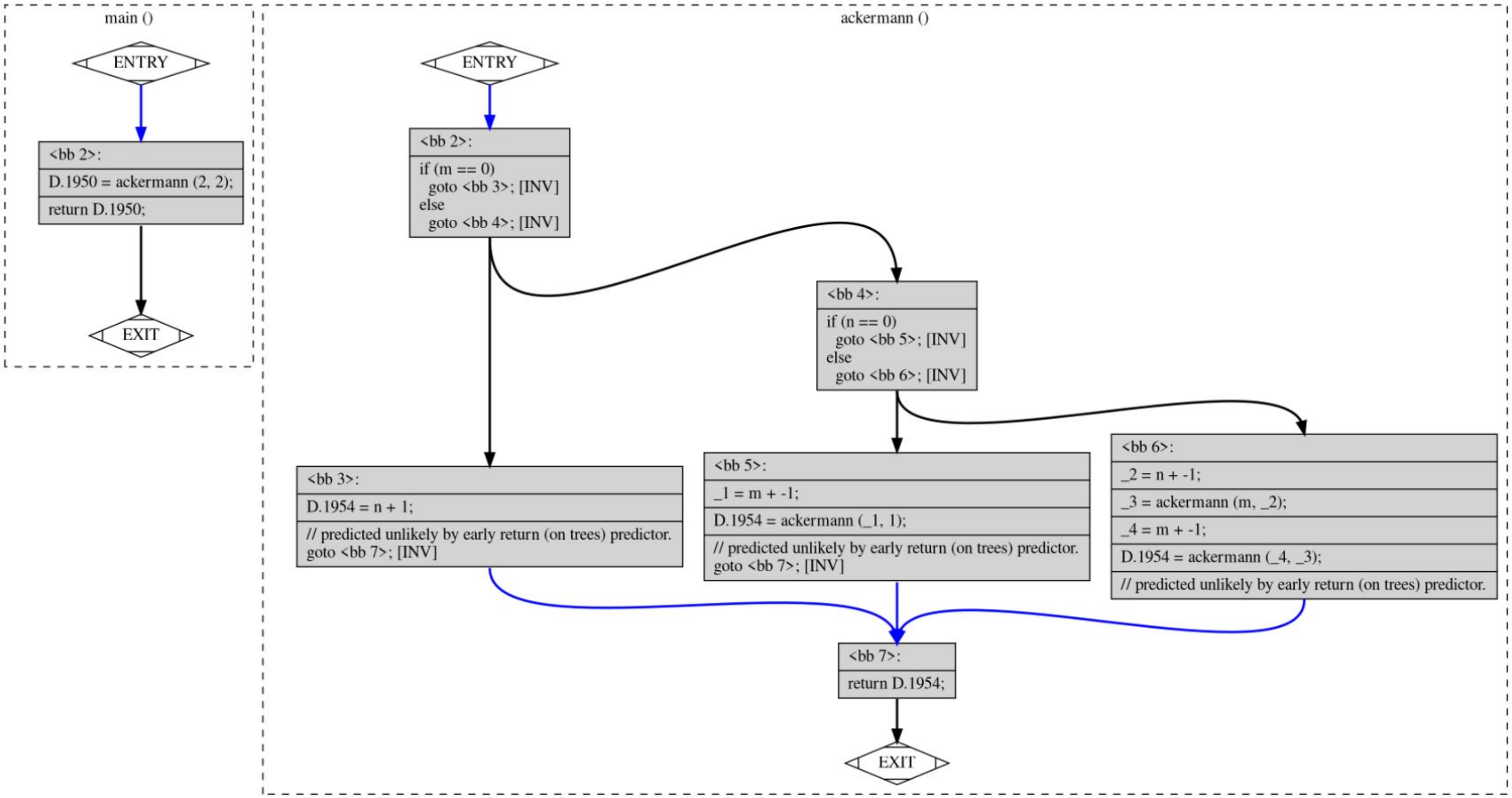
## GIMPL

```
int main ()  {
 int D.1950;
 {
   D.1950 = ackermann (2, 2);
   return D.1950;
 }
 D.1950 = 0;
 return D.1950;
}

int ackermann (int m, int n) {
 int D.1954;

 if (m == 0) goto <D.1952>; else goto <D.1953>;
 <D.1952>:
 D.1954 = n + 1;
 // predicted unlikely by early return (on trees) predictor.
 return D.1954;
 <D.1953>:
 if (n == 0) goto <D.1955>; else goto <D.1956>;
 <D.1955>:
 _1 = m + -1;
 D.1954 = ackermann (_1, 1);
 // predicted unlikely by early return (on trees) predictor.
 return D.1954;
 <D.1956>:
 _2 = n + -1;
 _3 = ackermann (m, _2);
 _4 = m + -1;
 D.1954 = ackermann (_4, _3);
 // predicted unlikely by early return (on trees) predictor.
 return D.1954;
}
```

## CONTROL FLOW GRAPH

```
;; Function main (main, funcdef_no=1,
decl_uid=1947, cgraph_uid=2, symbol_order=1)

Removing basic block 3
Merging blocks 2 and 4
;; 1 loops found
;;
;; Loop 0
;;  header 0, latch 1
;;  depth 0, outer -1
;;  nodes: 0 1 2
;; 2 succs { 1 }
int main ()
{
  int D.1950;

  <bb 2> :
  D.1950 = ackermann (2, 2);
  return D.1950;

}



;; Function ackermann (ackermann, funcdef_no=0,
```

# Ackermann CFG: Basic blocks definition




This plot has been made with the following commands:

**$> cd src/lab4**

**$> make ackermann_function.o graphs**

# Ackermann SSA:

**CONTROL FLOW GRAPH**

```
;; Function main (main, funcdef_no=1,
decl_uid=1947, cgraph_uid=2, symbol_order=1)

Removing basic block 3
Merging blocks 2 and 4
;; 1 loops found
;;
;; Loop 0
;;  header 0, latch 1
;;  depth 0, outer -1
;;  nodes: 0 1 2
;; 2 succs { 1 }
int main ()
{
  int D.1950;

  <bb 2> :
  D.1950 = ackermann (2, 2);
  return D.1950;

}



;; Function ackermann (ackermann, funcdef_no=0,
```

**SSA**

```
;; Function main (main, funcdef_no=1,
decl_uid=1947, cgraph_uid=2, symbol_order=1)

int main ()
{
  int _3;

  <bb 2> :
  _3 = ackermann (2, 2);
  return _3;

}



;; Function ackermann (ackermann, funcdef_no=0,
decl_uid=1944, cgraph_uid=1, symbol_order=0)

int ackermann (int m, int n)
{
  int _1;
  int _2;
  int _3;
  int _4;
  int _5;
  int _12;
```

# Optimization passes

There are a huge number of optimizations passes that follow.

As a simple tuning we can control it using the -O flag

| option | optimization level | execution time | code size | memory usage | compile time |
|---|---|---|---|---|---|
| **-O0** | optimization for compilation time (default) | + | + | - | - |
| **-O1 ( or -O )** | optimization for code size and execution time | - | - | + | + |
| **-O2** | optimization more for code size and execution time | -- | | + | ++ |
| **-O3** | optimization more for code size and execution time | --- | | + | +++ |
| **-Os** | optimization for code size | | -- | | ++ |
| **-Ofast** | O3 with fast math calculations non-standard | --- | | + | +++ |

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

# Type resolution & Inlining

```
__DEFINE_PRINT_LOG(log_error, LOG_ERROR)
```

```
static inline void log_error(const char *msg, ...) {
   va_list args;
   va_start (args, msg);
   print_log(msg, 2, args);
   va_end (args);
}
```

# Constant folding

While **count** and **total** change in the course of the program, **multiplied** does not: it starts off 0, and every time it is multiplied via multiplied = multiplied * count it remains 0.
**We can thus substitute 0 for it throughout the program:**

```
int main(int argc, char **argv){

-  int count = 0, total = 0, multiplied = 0;
+  int count = 0, total = 0;

   while(count < n){
     count += 1;
-     multiplied *= count;
-    if (multiplied < 100) log_info("count: %d\n",count);
+    if (0 < 100) log_info("count: %d\n",count);
     total += ackermann(2, 2);
-    total += ackermann(multiplied, n);
+    total += ackermann(0, n);
     int d1 = ackermann(n, 1);
-     total += d1 * multiplied;
     int d2 = ackermann(n, count);
     if (count % 2 == 0) total += d2;
   }
```

multiplied is actually a constant

it never changes during the program execution ...

As a consequence `d1 * multiplied` is always `0`, and thus `total += d1 * multiplied` does nothing, and can be removed

# Dead code Elimination

```
int main(int argc, char **argv){

   int count = 0, total = 0;

   while(count < n){
     count += 1;
     if (0 < 100) log_info("count: %d\n",count);
     total += ackermann(2, 2);
     total += ackermann(0, n);
-     int d1 = ackermann(n, 1);
     int d2 = ackermann(n, count);
     if (count % 2 == 0) total += d2;
   }
```

now d1 is useless for the program

# Branch Elimination

```
int main(int argc, char **argv){

   int count = 0, total = 0;

   while(count < n){
     count += 1;
     if (0 < 100) log_info("count: %d\n",count);
- if (0 < 100) log_info("count: %d\n",count);
+ log_info("count: %d\n",count);
     total += ackermann(2, 2);
     total += ackermann(0, n);
     int d2 = ackermann(n, count);
     if (count % 2 == 0) total += d2;
   }
```

now the branch has no sense

# Partial Evaluation

Let us take a look at the three remaining calls to ackermann function:

```
int main(int argc, char **argv){

   int count = 0, total = 0;

   while(count < n){
     count += 1;
     log_info("count: %d\n",count);
-    total += ackermann(2, 2);
+    total += 7
-    total += ackermann(0, n);
+    total += n + 1
     int d2 = ackermann(n, count);
     if (count % 2 == 0) total += d2;
   }
```

```
static int ackermann(int m, int n){
  if (m == 0) return n + 1;
  else if (n == 0) return ackermann(m - 1, 1);
  else return ackermann(m - 1, ackermann(m, n - 1));
  }
```

- The first has two constant arguments. We can see that the function is pure, and evaluate the function up-front to find **ackermann(2, 2)** must be equal to **7**
- The second has one constant argument **0**, and one unknown argument **n**. We can feed this into the definition of **ackermann**, and find that when **m** is **0**, the function **always returns n + 1**
- The third has two unknown arguments: n and count, and so there we leave it in place for now

# Late scheduling

Lastly, we can see that the definition of d2 only gets used in the if (count % 2 == 0) conditional. Since the computation of ackermann is pure, we can thus move that call into the conditional, so that **it doesn't get computed unless it actually gets used**:

```
int main(int argc, char **argv){

   int count = 0, total = 0;

   while(count < n){
     count += 1;
     log_info("count: %d\n",count);
     total += 7
     total += n + 1
-    int d2 = ackermann(n, count);
-    if (count % 2 == 0) total += d2;
+    if (count % 2 == 0) {
+      int d2 = ackermann(n, count);
+      total += d2;
+    }
   }
```

This lets us avoid half the calls to ackermann(n, count), speeding things up by not calling it when not necessary.

# Optimization - side by side comparison

CODING EXAMPLE CODING EXAMPLE

### ORIGINAL FRAGMENT

```c
int main(int argc, char **argv){
 int count = 0, total = 0, multiplied = 0;

 while(count < n){
   count += 1;
   multiplied *= count;
   if (multiplied < 100) log_info("count: %d\n",count);
   total += ackermann(2, 2);
   total += ackermann(multiplied, n);
   int d1 = ackermann(n, 1);
   total += d1 * multiplied;
   int d2 = ackermann(n, count);
   if (count % 2 == 0) total += d2;
 }
 return total;
}
```

### OPTIMIZED RESULT

```c
static int main(int n){
 int count = 0, total = 0;

 while(count < n){
   count += 1;

   log_info("count: %d\n",count);
   total += 7;
   total += n + 1;



   if (count % 2 == 0) {
     total += d2;
     int d2 = ackermann(n, count);
   }
 }
 return total;
}
```

# Ackermann RTL:

```
;; Function ackermann (ackermann, funcdef_no=0, decl_uid=1944, cgraph_uid=1, symbol_order=0)

ackermann

Dataflow summary:
;;  fully invalidated by EH         0 [ax] 1 [dx] 2 [cx] 4 [si] 5 [di] 8 [st] 9 [st(1)] 10 [st(2)] 11 [st(3)] 12 [st(4)] 13 [st(5)] 14 [st(6)] 15 [st(7)]
17 [flags] 18 [fpsr] 20 [xmm0] 21 [xmm1] 22 [xmm2] 23 [xmm3] 24 [xmm4] 25 [xmm5] 26 [xmm6] 27 [xmm7] 28 [mm0] 29 [mm1] 30 [mm2] 31 [mm3] 32 [mm4] 33
[mm5] 34 [mm6] 35 [mm7] 36 [r8] 37 [r9] 38 [r10] 39 [r11] 44 [xmm8] 45 [xmm9] 46 [xmm10] 47 [xmm11] 48 [xmm12] 49 [xmm13] 50 [xmm14] 51 [xmm15] 52
[xmm16] 53 [xmm17] 54 [xmm18] 55 [xmm19] 56 [xmm20] 57 [xmm21] 58 [xmm22] 59 [xmm23] 60 [xmm24] 61 [xmm25] 62 [xmm26] 63 [xmm27] 64 [xmm28] 65 [xmm29] 66
[xmm30] 67 [xmm31] 68 [k0] 69 [k1] 70 [k2] 71 [k3] 72 [k4] 73 [k5] 74 [k6] 75 [k7]
;;  hardware regs used      7 [sp]
;;  regular block artificial uses    6 [bp] 7 [sp]
;;  eh block artificial uses         6 [bp] 7 [sp] 16 [argp] 19 [frame]
;;  entry block defs        0 [ax] 1 [dx] 2 [cx] 4 [si] 5 [di] 6 [bp] 7 [sp] 19 [frame] 20 [xmm0] 21 [xmm1] 22 [xmm2] 23 [xmm3] 24 [xmm4] 25 [xmm5] 26
[xmm6] 27 [xmm7] 36 [r8] 37 [r9]
;;  exit block uses         0 [ax] 6 [bp] 7 [sp] 19 [frame]
;;  regs ever live          0 [ax] 1 [dx] 4 [si] 5 [di] 6 [bp] 7 [sp] 17 [flags]
;;  ref usage     r0={12d,10u} r1={6d,2u} r2={4d} r4={7d,4u} r5={7d,4u} r6={3d,19u} r7={4d,15u} r8={3d} r9={3d} r10={3d} r11={3d} r12={3d} r13={3d}
r14={3d} r15={3d} r17={9d,2u} r18={3d} r19={1d,1u,3e} r20={4d} r21={4d} r22={4d} r23={4d} r24={4d} r25={4d} r26={4d} r27={4d} r28={3d} r29={3d} r30={3d}
r31={3d} r32={3d} r33={3d} r34={3d} r35={3d} r36={4d} r37={4d} r38={3d} r39={3d} r44={3d} r45={3d} r46={3d} r47={3d} r48={3d} r49={3d} r50={3d} r51={3d}
r52={3d} r53={3d} r54={3d} r55={3d} r56={3d} r57={3d} r58={3d} r59={3d} r60={3d} r61={3d} r62={3d} r63={3d} r64={3d} r65={3d} r66={3d} r67={3d} r68={3d}
r69={3d} r70={3d} r71={3d} r72={3d} r73={3d} r74={3d} r75={3d}
;;    total ref usage 306{246d,57u,3e} in 33{30 regular + 3 call} insns.
(note 1 0 5 NOTE_INSN_DELETED)
(note 5 1 56 2 [bb 2] NOTE_INSN_BASIC_BLOCK)
(insn/f 56 5 57 2 (set (mem:DI (pre_dec:DI (reg/f:DI 7 sp)) [0  S8 A8])
        (reg/f:DI 6 bp)) "ackermann_function.c":1:35 52 {*pushdi2_rex64}
     (nil))
(insn/f 57 56 58 2 (set (reg/f:DI 6 bp)
        (reg/f:DI 7 sp)) "ackermann_function.c":1:35 74 {*movdi_internal}
     (nil))
(insn/f 58 57 59 2 (parallel [
            (set (reg/f:DI 7 sp)
                (plus:DI (reg/f:DI 7 sp)
                    (const_int -16 [0xfffffffffffffff0])))
            (clobber (reg:CC 17 flags))
            (clobber (mem:BLK (scratch) [0  A8]))
        ]) "ackermann_function.c":1:35 1143 {pro_epilogue_adjust_stack_add_di}
```

AND MANY MORE ...

# Help the compiler !

# Help the compiler: **1 - Inline functions**

So we saw that understand the compile things are becoming difficult !

So at a simple writing practice what can we do to help the compiler perform its magic ?

One possibility is to avoid function call when not necessary …

```
#define __DEFINE_PRINT_LOG(name, errno) \
static inline void name(const char *msg, ...) { \
   va_list args; \
   va_start (args, msg); \
   print_log(msg, errno, args); \
   va_end (args); \
}
__DEFINE_PRINT_LOG(log_fatal, LOG_FATAL_ERROR)
__DEFINE_PRINT_LOG(log_error, LOG_ERROR)
__DEFINE_PRINT_LOG(log_warning, LOG_WARNING)
__DEFINE_PRINT_LOG(log_debug, LOG_DEBUG)
__DEFINE_PRINT_LOG(log_info, LOG_INFO)
#undef __DEFINE_PRINT_LOG
```

The inline attribute suggest the compiler to put the function body in place of the actual function call.. just appending the function subtree on the call node.

# Help the compiler: **1 - Inline functions**

```
inline return-type function-name(parameters) {
    // function code
}
```

## Why doing that ?

**Inline functions provide following advantages:**

1) Function call overhead doesn't occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

# Help the compiler: **1 - Inline functions**

Remember, **inlining is only a request to the compiler, not a command**. Compiler can ignore the request for inlining.

**Compiler may not perform inlining in such circumstances like:**

1) If a function contains a **loop**. (for, while, do-while)

2) If a function contains **static variables**.

3) If a **function is recursive**.

4) If a function return type is other than void, and the **return statement doesn't exist** in function body.

5) If a function contains **switch or goto** statement.

## Did our function get inlined ??

```
gcc -O3 -fopt-info-missed=missed.all
```

outputs missed optimization report from all the passes into missed.all.

```
gcc -O3 -fopt-info-inline-optimized-missed=inline.txt
```

will output information about missed optimizations as well as optimized locations from all the inlining passes into inline.txt.

# Help the compiler: **1 - Inline functions**

**ackermann.c:65:15: missed: Not inlining: recursive call.**
**ackermann.c:65:15: missed: Not inlining: recursive call.**
**ackermann.c:64:27: missed: Not inlining: recursive call.**

```
60
61  // https://en.wikipedia.org/wiki/Ackermann_function
62  static int ackermann(int m, int n){
63    if (m == 0) return n + 1;
64    else if (n == 0) return ackermann(m - 1, 1);
65    else return ackermann(m - 1, ackermann(m, n - 1));
66  }
67
```

ackermann.c:42:5: optimized: Inlining vprintf/0 into print_log/23 (a
ackermann.c:54:1: optimized: Inlining print_log/23 into log_error/2
ackermann.c:57:1: optimized: Inlining print_log/23 into log_info/28
ackermann.c:74:11: optimized: Inlining atoi/11 into main/30 (always_inline).
ackermann.c:85:14: missed: will not early inline: main/30->ackermann/29, growth 8 exceeds --param early-inlining-insns divided by number of calls
ackermann.c:83:14: missed: will not early inline: main/30->ackermann/29, growth 8 exceeds --param early-inlining-insns divided by number of calls
ackermann.c:82:14: missed: will not early inline: main/30->ackermann/29, growth 8 exceeds --param early-inlining-insns divided by number of calls
ackermann.c:81:14: missed: will not early inline: main/30->ackermann/29, growth 8 exceeds --param early-inlining-insns divided by number of calls
ackermann.c:54:1: missed: not inlinable: log_error.constprop/44 -> __builtin_va_start/32, function body not available
ackermann.c:54:1: missed: not inlinable: log_error.constprop/44 -> __builtin_va_end/33, function body not available
ackermann.c:57:1: missed: not inlinable: log_info.constprop/43 -> __builtin_va_start/32, function body not available
ackermann.c:57:1: missed: not inlinable: log_info.constprop/43 -> __builtin_va_end/33, function body not available
[ … ]

```
46  #define __DEFINE_PRINT_LOG(name, errno) \
47  static inline void name(const char *msg, ...) { \
48      va_list args; \
49      va_start (args, msg); \
50      print_log(msg, errno, args); \
51      va_end (args); \
52  }
53  __DEFINE_PRINT_LOG(log_fatal, LOG_FATAL_ERROR)
54  __DEFINE_PRINT_LOG(log_error, LOG_ERROR)
55  __DEFINE_PRINT_LOG(log_warning, LOG_WARNING)
56  __DEFINE_PRINT_LOG(log_debug, LOG_DEBUG)
57  __DEFINE_PRINT_LOG(log_info, LOG_INFO)
58  #undef __DEFINE_PRINT_LOG
```

# Help the Hardware

# Memory “locality”

The ***locality of reference***, also known as the ***principle of locality***, is the tendency of a processor to access the same set of memory locations repetitively over a short period of time.

Systems that exhibit strong locality of reference are great candidates for performance optimization: caching, prefetching for memory and advanced branch predictors at the pipelining stage of a processor core.

- **Temporal locality**: If at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. There is temporal proximity between adjacent references to the same memory location. In this case it is common to make efforts to store a copy of the referenced data in faster memory storage, to reduce the latency of subsequent references. Temporal locality is a special case of spatial locality (see below), namely when the prospective location is identical to the present location.

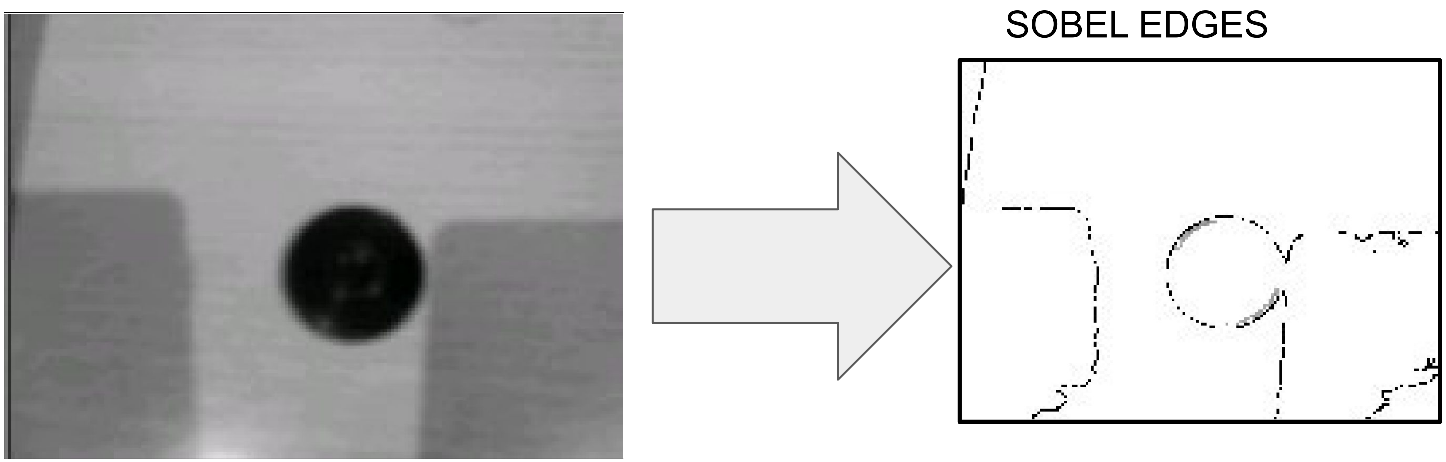
- **Spatial locality**: If a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access for subsequent reference.
    - **Memory locality** (or *data locality*): Spatial locality explicitly relating to memory.

- **Branch locality**: If there are only a few possible alternatives for the prospective part of the path in the spatial-temporal coordinate space. This is the case when an instruction loop has a simple structure, or the possible outcome of a small system of conditional branching instructions is restricted to a small set of possibilities. Branch locality is typically not spatial locality since the few possibilities can be located far away from each other.

- **Equidistant locality**: Halfway between spatial locality and branch locality. Consider a loop accessing locations in an equidistant pattern, i.e., the path in the spatial-temporal coordinate space is a dotted line. In this case, a simple linear function can predict which location will be accessed in the near future.

# Example: the Sobel filter

In the last lesson ( Lab3 ) we saw how to find edges in images by a convolutional filter with a proper 3x3 matrix, called the Sobel product.



# Sobel NonOptimized for locality

CODING EXAMPLE

CODING EXAMPLE

```
/* Sobel matrixes */
static const int GX[3][3] = {
        1, 0, -1,
        2, 0, -2,
        1, 0, -1
    };

static const int GY[3][3] = {
         1,  2,  1,
         0,  0,  0,
        -1, -2, -1
    };

static inline int _abs(int x) { return (x>0)?x:-x; }

/* Sobel Filter computation for Edge detection. */
int makeBorderNonOptimized(unsigned char *image, unsigned char *border, int cols, int rows, int threshold)
/* Input image is passed in the byte array image (cols x rows pixels)
   Filtered image is returned in byte array border */
{
    int x,y, i, j, sumX, sumY, sum;
    int numBlackPixels = 0;
```

# Sobel NonOptimized for locality

CODING EXAMPLE

CODING EXAMPLE

```
for(y = 0; y < rows; ++y)    {
    for(x = 0; x < cols; ++x)    {
        sumX = 0;
        sumY = 0;
        /* handle image boundaries */
        if(y == 0 || y == rows-1) sum = 0;
        else if(x == 0 || x == cols-1) sum = 0;

        /* Convolution starts here */
        else {
            /* X Gradient */
            for(i = -1; i <= 1; i++)
                for(j =- 1; j <= 1; j++)
                    sumX += (int)(image [ x + i + (y + j)*cols]) * GX[i+1][j+1];

            /* Y Gradient */
            for(i = -1; i <= 1; i++)
                for(j =- 1; j <= 1; j++)
                    sumY += (int)(image [ x + i + (y + j)*cols]) * GY[i+1][j+1];

            /* Gradient Magnitude approximation to avoid square root operations */
            sum = _abs(sumX) + _abs(sumY);
        }
```

# Sobel NonOptimized for locality

```
            if(sum > 255) sum = 255;
            if(sum < threshold) sum = 0;
            else
                ++numBlackPixels;
            border[x + y*cols] = 255 - (unsigned char)(sum);
        }
    }
    return numBlackPixels;
}
```


CODING EXAMPLE

# Sobel Optimized for locality

CODING EXAMPLE

CODING EXAMPLE

```
int makeBorderOptimized(unsigned char *image, unsigned char *border, int cols, int rows, int threshold) {
    int x = 0, y, sumX, sumY, sum, numBlackPixels = 0;
    /* Variables to hold the 3x3 portion of the image used in the computation
       of the Sobel filter output */
    int c11,c12,c13,c21,c22,c23,c31,c32,c33;
    for(y = 0; y <= (rows-1); y++) {
        /* First image row: the first row of cij is zero */
        if(y == 0)  c11 = c12 = c13 = 0;
        else {
        /* First image column: the first column of cij matrix is zero */
            c11=0;
            c12 = *(image + (y - 1) * cols);
            c13 = *(image + 1 + (y - 1)*cols);
        }
        c21 = 0;
        c22 = *(image + y*cols);
        c23 = *(image + 1 + y*cols);
        if(y == rows - 1) {
            /* Last image row: the third row of cij matrix is zero */
            c31 = c32 = c33 = 0;
        }
        else {
            c31=0;
            c32 = *(image + (y + 1)*cols);
            c33 = *(image + 1 + (y + 1)*cols);
        }
```

# Sobel Optimized for locality


CODING EXAMPLE

```
/* The 3x3 matrix corresponding to the first pixel of the current image
   row has been loaded in program variables.
   The following iterations will only load
   from memory the rightmost column of such matrix */
        for(x = 0; x <= (cols-1); x++) {
            sumX = sumY = 0;
             /* Skip image boundaries */
            if(y == 0 || y == rows-1)  sum = 0;
            else if(x == 0 || x == cols-1)  sum = 0;
             /* Convolution starts here.
                GX and GY parameters are now "cabled" in the code */
            else  {
                sumX = sumX - c11;
                sumY = sumY + c11;
                sumY = sumY + 2*c12;
                sumX = sumX + c13;
                sumY = sumY + c13;
                sumX = sumX - 2 * c21;
                sumX = sumX + 2*c23;
                sumX = sumX - c31;
                sumY = sumY - c31;
                sumY = sumY - 2*c32;
                sumX = sumX + c33;
                sumY = sumY - c33;
                sum = _abs(sumX) + _abs(sumY);
            }
```

# Sobel Optimized for locality

CODING EXAMPLE

```
/* Move one pixel on the right in the current row.
   Update the first/last row only if not in the first/last image row */
            if(y > 0) {
                c11 = c12;
                c12 = c13;
                c13 = *(image + x + 2 + (y - 1) * cols);
            }
            c21 = c22;
            c22 = c23;
            c23 = *(image + x +2 + y * cols);
            if(y < cols - 1)  {
                c31 = c32;
                c32 = c33;
                c33 = *(image + x + 2 + (y + 1) * cols);
            }
            if(sum > 255) sum = 255;
            if(sum < threshold)
              sum=0;
            else
              numBlackPixels++;
            /* Report the new pixel in the output image */
            *(border + x + y*cols) = 255 - (unsigned char)(sum);
        }
    }
    return numBlackPixels;
}
```

CODING EXAMPLE

# Analisi sperimentale

Non possiamo utilizzare un cronometro esterno, il tempo misurato comprenderebbe:

- Operazioni svolte dal sistema operativo
- Operazioni svolte da altre applicazioni
- Tempo per leggere input / scrivere output

I programmi non hanno una conoscenza reale del tempo, per loro il tempo e' relativo alla effettiva esecuzione dei cicli. Per ottenere dei valori corretti e' necessario fare delle richieste al sistema operativo.  Per questo esistono funzioni specifiche:  **time.h**

La funzione **clock()** restituisce il numero di cicli macchina consumati dal processo

La funzione **clock_gettime()** restituisce il tempo trascorso

# Esempio: la funzione clock_gettime()

```c
#include <stdio.h>
#include <time.h>   // for clock_t, clock()
#include <unistd.h> // for sleep()

#define BILLION  1000000000.0

// main function to find the execution time of a C program
int main()
{
   struct timespec start, end;

   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);

   // do some stuff here
   sleep(3);

   clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);

   // time_spent = end - start
   double time_spent = (end.tv_sec - start.tv_sec) +
                 (end.tv_nsec - start.tv_nsec) / BILLION;

   printf("Time elpased is %f seconds", time_spent);
   return 0;
}
```

```c
struct timespec {
    time_t    tv_sec;
    long      tv_nsec;
}
```

**CLOCK_REALTIME** riporta l'ora effettiva dell'orologio di sistema.

**CLOCK_MONOTONIC** serve per misurare il tempo reale relativo. Avanza alla stessa velocità del flusso di tempo effettivo, ma non è soggetto a discontinuità delle regolazioni manuali o automatiche (NTP) all'orologio di sistema.

**CLOCK_PROCESS_CPUTIME_ID** serve per misurare la quantità di tempo della CPU consumato dal processo.

**CLOCK_THREAD_CPUTIME_ID** serve per misurare la quantità di tempo CPU consumato dal thread. È supportato dai kernel moderni e glibc dal 2.6.12.

# Example: the Sobel filter

Example of Soblel filtering benchmark

cd src/lab4

[crtp@osboxes lab4]$ ./test_gradient test_image.pix test_image_borders.pix **0**
Rows: 512 Cols: 512
Num Pixels: 262144 Num Black : 44777
Elapsed system time(us) 2507
Elapsed process time(us): 2523.152000

NonOptimized

[crtp@osboxes lab4]$ ./test_gradient test_image.pix test_image_borders.pix **1**
Rows: 512 Cols: 512
Num Pixels: 262144 Num Black : 44777
Elapsed system time(us) 3765
Elapsed process time(us): 1697.504000

Optimized

## EXERCISE 1

Apply process CPU time computation to the **v4l_example** application seen in **lab3**

A microsecond value of the findBorder function can be printed on console during frame acquisition.

## EXERCISE 2

Acquire timing statistics for many repeated attempt to find borders on **test_gradients** application in **lab4**.

The collated results can be plotted with different histograms showing the distributions of timing for different Optimization flags ( -O0, -O1, -O2 and -O3 ).