High Performance Co$^{\textit{mputing}}_{\textit{ding}}$ (HPC)

Alberto Dassatti - 2024

Do you need it?

COMPILER

1. Using higher level languages improves abstraction and productivity

Compiler

1. Using higher level languages improves abstraction and productivity
2. Using higher level languages improves portability

Compiler

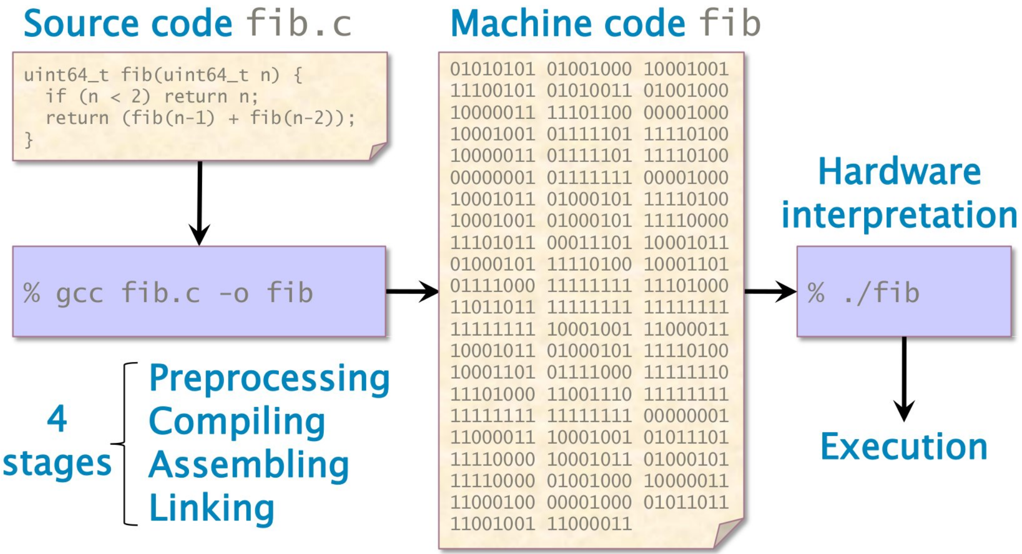
1. Using higher level languages improves abstraction and productivity
2. Using higher level languages improves portability
3. Compilers translate your code (assembly is complex)

Compiler

1. Using higher level languages improves abstraction and productivity
2. Using higher level languages improves portability
3. Compilers translate your code (assembly is complex)
4. Compilers optimize the code during the translation

Compiler

1. Using higher level languages improves abstraction and productivity
2. Using higher level languages improves portability
3. Compilers translate your code (assembly is complex)
4. Compilers optimize the code during the translation
5. Compilers embed many many many years of programming experience


Source code fib.c
uint64_t fib(uint64_t n) {
  if (n < 2) return n;
  return (fib(n-1) + fib(n-2));
}
% gcc fib.c -o fib
4 stages
Preprocessing
Compiling
Assembling
Linking
Machine code fib
01010101 01001000 10001001
11100101 01010011 01001000
10000011 11101100 00001000
10001001 01111101 11110100
10000011 01111101 11110100
00000001 01111111 00001000
10001011 01000101 11110100
10001001 01000101 11110000
11101011 00011101 10001011
01000101 11110100 10001101
01111000 11111111 11101000
11011011 11111111 11111111
11111111 10001001 11000011
10001011 01000101 11110100
10001101 01111000 11111110
11101000 11001110 11111111
11111111 11111111 00000001
11000011 10001001 01011101
11110000 10001011 01000101
11110000 01001000 10000011
11000100 00001000 01011011
11001001 11000011
Hardware interpretation
% ./fib
Execution

Source, machine, & assembly

Binary executable
fib with debug symbols

```
% objdump -S fib
```

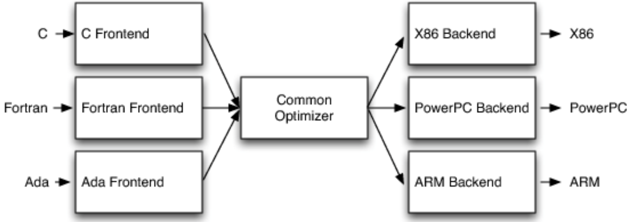
```
uint64_t fib(uint64_t n) {
  4004f0: 55                  push   %rbp
  4004f1: 48 89 e5            mov    %rsp,%rbp
  4004f4: 53                  push   %rbx
  4004f5: 48 83 ec 18         sub    $0x18,%rsp
  4004f9: 48 89 7d f0         mov    %rdi,-0x10(%rbp
  if (n < 2) return n;
  4004fd: 48 83 7d f0 01      cmpq   $0x1,-0x10(%rbp)
  400502: 77 0a               ja     40050e <fib+0x1e>
  400504: 48 8b 45 f0         mov    -0x10(%rbp),%rax
  400508: 48 89 45 e8         mov    %rax,-0x18(%rbp)
  40050c: eb 24               jmp    400532 <fib+0x42>
  return (fib(n-1) + fib(n-2));
  40050e: 48 8b 45 f0         mov    -0x10(%rbp),%rax
  400512: 48 8d 78 ff         lea    -0x1(%rax),%rdi
  400516: e8 d5 ff ff ff      callq  4004f0 <fib>
  40051b: 48 89 c3            mov    %rax,%rbx
  40051e: 48 8b 45 f0         mov    -0x10(%rbp),%ra
  400522: 48 8d 78 fe         lea    -0x2(%rax),%rdi
  400526: e8 c5 ff ff ff      callq  4004f0 <fib>
  40052b: 48 01 c3            add    %rax,%rbx
  40052e: 48 89 5d e8         mov    %rbx,-0x18(%rbp
  400532: 48 8b 45 e8         mov    -0x18(%rbp),%rax
}
  400536: 48 83 c4 18         add    $0x18,%rsp
  40053a: 5b                  pop    %rbx
  40053b: c9                  leaveq
  40053c: c3                  retq
```

source

What a compiler is

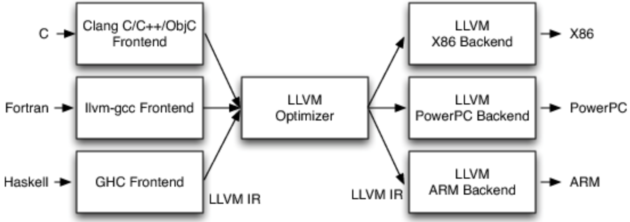
source

What a compiler is



source

What a compiler is



source

Front end

Translate the high level input language in the compiler internal rapresentation.

LLVM IR (intermidiate rappresentation)

```bash
[al@lap ~]# clang file.c -S emit-llvm
```

GCC has two IR, GIMPLE for optimization and RTL for low level finalization.

```bash
[al@lap ~]# gcc file.c -fdump-tree-gimple
```

BACK END

From Wikipedia: The main phases of the back end include the following:

- Analysis: This is the gathering of program information from the intermediate
  …
- Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms.
- Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes (see also Sethi-Ullman algorithm). Debug data may also need to be generated to facilitate debugging.

Optimization

A compiler has two goals: creating machine code from a higer level language and optimizing the produced code.

bash

```bash
[al@lap ~]# gcc file.c -OOPT_LEVEL
```

For a `gcc` specifc description of `OPT_LEVEL` look here. Most optimization can be activated individually with `-fname`.

Time to time it is interesting to see what the compiler is doing. Here a list of useful options in the section `-fopt-info`.

OPTIMIZATION

If you haven't compilded `gcc` yourself, or you forgot how it was configured, this command is very useful

```bash
[al@lap ~]# gcc -O2 -Q --help=optimizers
```

Optimization

Some low level optimizations are machine dependent, so it is a good practice to specify the target (even if native).

```bash
[al@lap ~]# gcc file.c -O3 -march=native -mSSE3 -m64
```

All available options are listed in the doc.

Performance difference is *huge*. Always use the optimizer. Many compilers defaults to no optimization (`gcc` among them).

If cross-compiling for ARM there are a lot of specific options you can activate (link).

LIMITATIONS

A compiler has to prove the modification done has no side effects, whenever this prove is impossible, the corresponding optimization is disabled.

- If in doubt, the compiler is conservative.
- Must not change program behavior under any possible condition.
- Most analysis are performed only within procedures.
- Most analysis are based only on static information.

INLINE EXPANSION OR INLINING

Replace function call with body of function.

Example

```
int add (int x, int y){
  return x + y;
}

int sub (int x, int y){
  return add (x, -y);
}
```

INLINE EXPANSION OR INLINING

Replace function call with body of function.

Example

```
int add (int x, int y){
  return x + y;
}

int sub (int x, int y){
  return add (x, -y);
}
```

Example

```
int sub (int x, int y){
  return x - y;
}
```

Usually requires source visibility.

Hard to force. Activated from `-O2`. Attribute `always_inline` may help.

DEAD CODE ELIMINATION

In compiler theory, **dead code elimination** (also known as DCE, **dead code** removal, **dead code** stripping, or **dead code** strip) is a compiler optimization to remove **code** which does not affect the program results.

Dead code elimination - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/**Dead_code_elimination**

Example

```
int global;
void f (){
  int i;
  i = 1;          /* dead store */
  global = 1;     /* dead store */
  global = 2;
  return;
  global = 3;     /* unreachable */
}
```

DEAD CODE ELIMINATION

In compiler theory, **dead code elimination** (also known as DCE, **dead code** removal, **dead code** stripping, or **dead code** strip) is a compiler optimization to remove **code** which does not affect the program results.

Dead code elimination - Wikipedia, the free encyclopedia
https://en.wikipedia.org/wiki/**Dead_code_elimination**

Example

```
int global;
void f (){
  int i;
  i = 1;          /* dead store */
  global = 1;     /* dead store */
  global = 2;
  return;
  global = 3;     /* unreachable */
}
```

Example

```
int global;
void f (){
  global = 2;
  return;
}
```

CONSTANT PROPAGATION OR FOLDING

Example

```
#include <stdio.h>

#define val 10

int main(){

int b = 100;
int k = (5 * val) + b;

return k;
}
```

CONSTANT PROPAGATION OR FOLDING

bash

```
[al@lap ~]# gcc -g -O3 constant.c && objdump -S a.out
```

```
Disassembly of section .text:

0000000000400400 <main>:
int b = 100;
int k = (5 * val) + b;

return k;
}
400400: b8 96 00 00 00          mov    $0x96,%eax
400405: c3                      retq
400406: 66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
40040d: 00 00 00
```

Output

UNROLLING

Example

```
#pragma GCC push_options
#pragma GCC optimize ("unroll-loops")

void add5(int a[20]) {
  int i;
    for(i=0; i < 20; i++) {
        a[i] += 5;
    }
}
#pragma GCC pop_options
```

Code Motion

Example

```
void set_row(double *a, double *b,int i, int n){
  int j;
  for (j = 0; j < n; j++)
    a[n*i+j] = b[j];
}
```

Code Motion

Example

```
void set_row(double *a, double *b,int i, int n){
  int j;
  for (j = 0; j < n; j++)
    a[n*i+j] = b[j];
}
```

Example

```
void set_row(double *a, double *b,int i, int n){
  int j;
  int ni = n*i;
  for (j = 0; j < n; j++)
   a[ni+j] = b[j];
}
```

Strength Reduction

Exchange operations with faster ones.

- `a*2` in `a << 1;`
- `a/4` in `a >> 2;`
- `a*5` in

Strength Reduction

Exchange operations with faster ones.

- `a*2` in `a << 1;`
- `a/4` in `a >> 2;`
- `a*5` in `(a << 2) + a;`

Share Common Subexpressions

Example

```
up    = val[(i-1)*n +j];
down  = val[(i+1)*n +j];
left  = val[i*n +j -1];
right = val[i*n +j+1];
sum = up + down + left + right;
```

Example

```
int inj = i*n + j;
up    = val[inj - n];
down  = val[inj + n];
left  = val[inj - 1];
right = val[inj + 1];
sum   = up + down + left + right;
```

Warnings

Example

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

Warnings

Example

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

Example

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

Example

```c
void sum_rows1(double *a, double *b, int n) {
  int i, j;
  for (i =0; i<n; i++) {
    b[i] = 0;
    for (j= 0; j < n; j++)
      b[i] += a[i*n + j];
  }
}
```

Example

```c
void sum_rows2(double *a, double *b, int n) {
  int i, j;
  for (i =0; i<n; i++) {
    double val = 0;
    for (j= 0; j < n; j++)
      val += a[i*n + j];
    b[i] = val;
  }
}
```

This optimization does not happen; Why?

Memory Aliasing

Compilers often cannot prove there is no aliasing.

- do the work by hand storing the value in a local variable.
- modern compiler can add run time checks.

Example

```
if (a + n < b || b + n < a)
/* further optimizations may be possible now */
...
else
/* aliased case */
```

- use some flags (`-fno-alias`, `-fargument-noalias`)
- use `restrict` keyword and compiler flag `-restrict`.

Vectorization

Disabled by default, regardless of optimization level.

```bash
[al@lap ~]# gcc ftree-vectorize -O2
```

SSE by default, for AVX

```bash
[al@lap ~]# gcc -mavx -march=corei7-avx
```

for a vectorization report

```bash
[al@lap ~]# gcc -ftree-vectorizer-verbose
```

Advanced Optimizations

IPO Interprocedural optimization

```bash
[al@lap ~]# gcc -fwhole-program --combine
```

LTO Link Time Optimization

```bash
[al@lap ~]# gcc -flto
```

PGO Profile-guided optimization

```bash
[al@lap ~]# gcc -fprofile-generate
[al@lap ~]# ./a.out
[al@lap ~]# gcc -fprofile-use
```

__BUILT_IN

Every compiler has a large number of internally available functions providing the access to optimized assembler routines or instrunctions.

Gcc documents its built-ins.

Ask the compiler

Modern compilers will give you visibility on their internal reasoning if asked kindly

```bash
[al@lap ~]# clang -O3 fsave-optimization-record -c foo.c
[al@lap ~]# utils/opt-viewer/opt-viewer.py foo.opt.yaml html
[al@lap ~]# open html/foo.c.html
```

Check out Remarks documentation for more details.

```bash
[al@lap ~]# gcc -fopt-info-options=missed
```

Here all the details.

SANITIZERS

The option `-fsanitize=ZZZ` is incredibly powerful. Not really for performance, but before making a code performant, you have to make it correct.

-fsanitize=address fast memory error detector.

-fsanitize=thread fast data race detector.

-fsanitize=undefined fast undefined behavior detector.

and many more https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html

Static Analysis

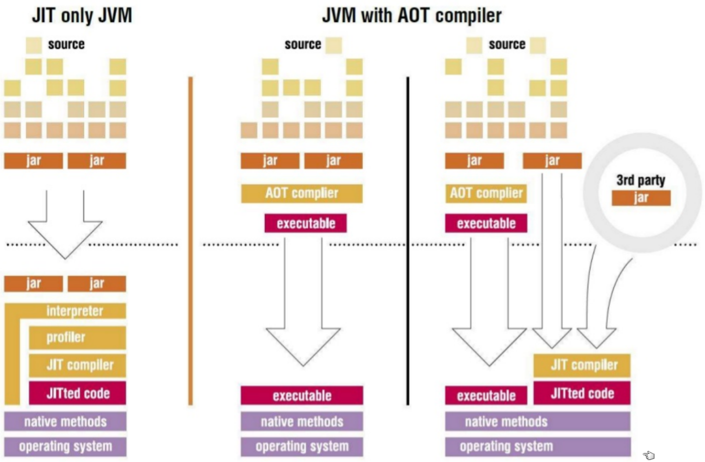
And why not checking for even more errors??

```bash
[al@lap ~]# gcc -fanalyzer      [1]
```

```bash
[al@lap ~]# scan-build make     [2]
```

[1] https://gcc.gnu.org
[2] https://clang-analyzer.llvm.org

Just In Time (JIT)



source

Ahead-of-Time (AOT) vs. JIT

- StackExchange
- Article
- One more

Polyhedral loop optimizers

- pluto
- Polly

How to check if the compiler did well: disassembly and assembly generation

```bash
[al@lap ~]# objdump -S a.out
```

Tooling is the future

Research on this front is never ending.

Some projects I think we should monitor closely:

- DACE from ETHZ
- JAX from google
- Taichi from MIT
- Halide from MIT
- Legion from Stanford

References

- How to Write Fast Numerical Code, Prof. Markus Püschel, ETHZ.
- Performance Engineering of Software Systems, Prof. Saman Amarasinghe, MIT.
- Optimizations with examples.
- Examples.
- Compiler Explorer.

Questions

?