NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 16: Introduction to optimizations

Michael Engel

Overview

- Optimizations
  - Definition, objectives, location in the compiler tool flow
  - Obtaining and applying evaluation criteria
  - Common vs. worst case
  - Optimization properties

Optimization

- What do we mean when we talk about an **optimizing** compiler?

- ***Mathematical*** optimization is the selection of a best element (with regard to some criterion) from some set of available alternatives

- With software, it is often hard to find a real optimum
    - Compiler "optimizations" **try** to minimize or maximize some attributes of an executable program
    - Large search space makes finding the real optimum impossible in many cases
    - In general, optimization is undecidable, often NP-complete
- Nevertheless, we will continue using the term "optimizations" here

Why optimization?

- To help programmers…
  - They (try to…) write modular, clean, high-level programs
- Compiler generates efficient, high-performance assembly
  - Programmers don’t write optimal code
- High-level languages make avoiding redundant computation inconvenient or impossible
  - e.g. `A[i][j] = A[i][j] + 1`
- Architectural independence
  - Optimal code depends on features not expressed to the programmer
  - Modern architectures assume optimization
- Important: **Ensure safety of optimizations**
  - Optimizations must not change the meaning (semantics) of a program!

Why optimization?

Code generated from simple AST traversal (+IR transformation) is often quite inefficient

```
int foo(int w) {
  int x, y, z;
  x = 3 + 5;
  y = x * w;
  z = y - 0;
  return z * 4;
}
```

gcc -O0

```
        .globl _foo
_foo:
LFB0:
        pushq   %rbp
LCFI0:
        movq    %rsp, %rbp
LCFI1:
        movl    %edi, -20(%rbp)
        movl    $8, -4(%rbp)
        movl    -4(%rbp), %eax
        imull   -20(%rbp), %eax
        movl    %eax, -8(%rbp)
        movl    -8(%rbp), %eax
        movl    %eax, -12(%rbp)
        movl    -12(%rbp), %eax
        sall    $2, %eax
        popq    %rbp
LCFI2:
        ret
```

gcc -O3

```
        .globl _foo
_foo:
LFB0:
        movl    %edi, %eax
        sall    $5, %eax
        ret
```

Optimization objectives

Which optimizations can a compiler try to achieve (examples)?

- Reduce **runtime** (in seconds)
- Reduce **code size** (in bytes)
- Reduce **power** consumption (in Watt)
- Reduce **energy** consumption (in Joule/Wh)
  - Objectives other than runtime relevant in embedded systems
- We also call all these objectives "**non-functional properties**"
  - They do not change the **semantics** of the code, but properties that influence its execution
- Code optimizations consist of two general stages:
  - **Analysis**: find optimization opportunities
  - **Transformation**: apply code changes

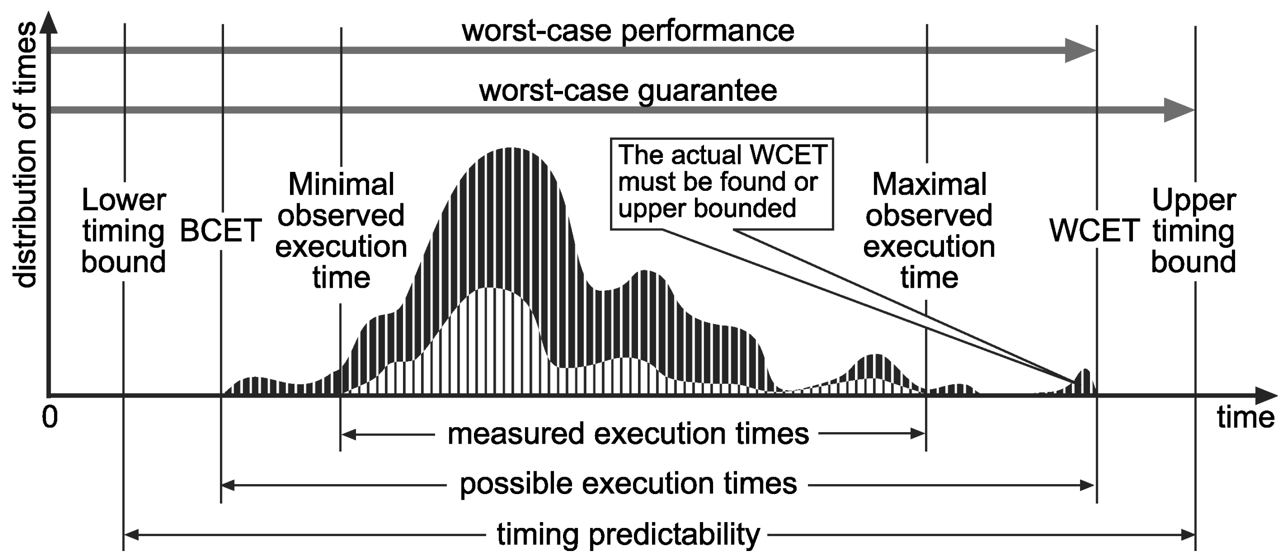
Optimizations… for what?

Most compiler optimizations consider the **common case**

- optimize cases providing largest benefit for the **average use case**

Some applications require optimization for the **worst case**

- in real-time systems, the **worst-case execution time** (WCET) determines if a system can operate safely under given real-time constraints
- a system that reacts too late can cause a catastrophe
- think of airbag controls in a car



[Wilhelm+08]

WCET: Worst-Case Execution Time
BCET: Best-Case Execution Time
ACET: Average-Case Execution Time

Optimizations become more difficult

Many architectural issues to think about

- Exploiting parallelism
    - instruction-level (ILP), thread, multi-core, accelerators
- Effective management of memory hierarchy
    - Registers [1], Caches (L1, L2, L3), Memory/NUMA, Disk
- Energy modes and heterogeneous multicores
    - Dynamic voltage-frequency scaling (DVFS), clock gating, big.LITTLE architectures

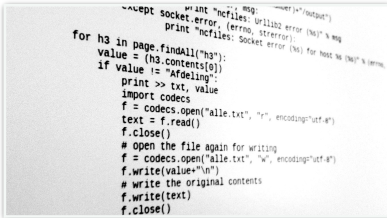
Small architectural changes have big impact – hard to reason about

Example

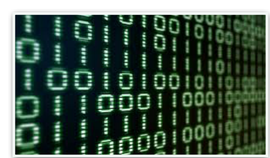
- Program optimised for CPU with Random cache replacement
- What do you change for new machine with LRU?

Where to apply optimizations

Source code



Lexical analysis → Syntax analysis → Semantic analysis → IR generation → *IR* → **IR optimization** (*IR*) → *IR* → Code generation → *machine code* → **Code optimization** →



machine-level program

Many analyses and transformations are general (not dependent on the target machine), so they can be easily applied on the **IR level**

Some analyses and optimizations are machine-dependent and better applied on the **machine code level**

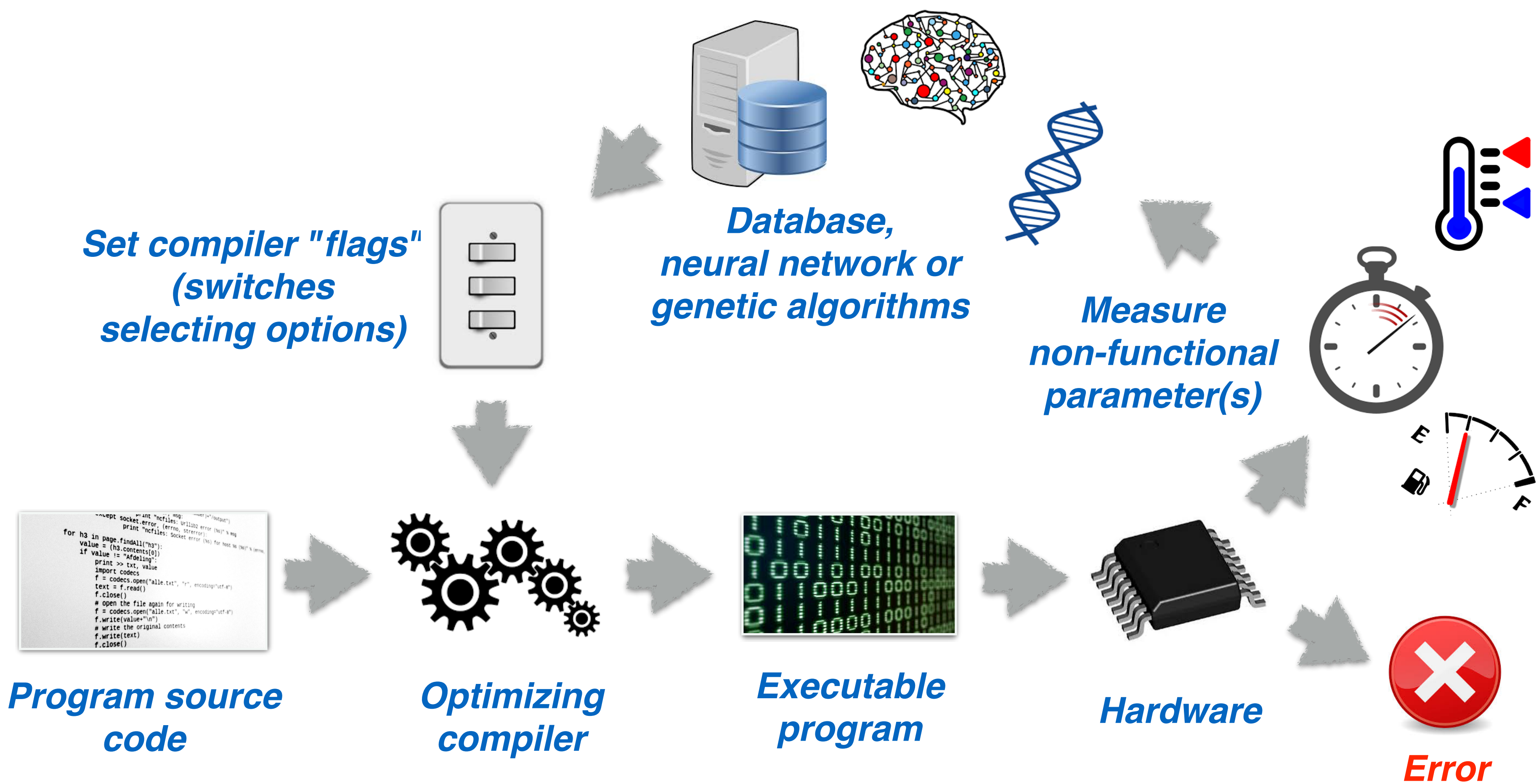
Optimization approaches

How can a compiler know that a **transformation** actually leads to an **optimization?**

- Simple approach: **hope for the best**
  - Example: "a lower number of instruction results in faster code"
  - This has worked surprisingly well for early architectures
- **Apply heuristics**
  - Used in many optimization decisions when concrete data or models are not available or search space too large
  - Examples:
    - Inlining decisions, Unrolling decisions, Packed-data (SIMD) optimization decisions, Instruction selection, Register allocation, Instruction scheduling, Software pipelining

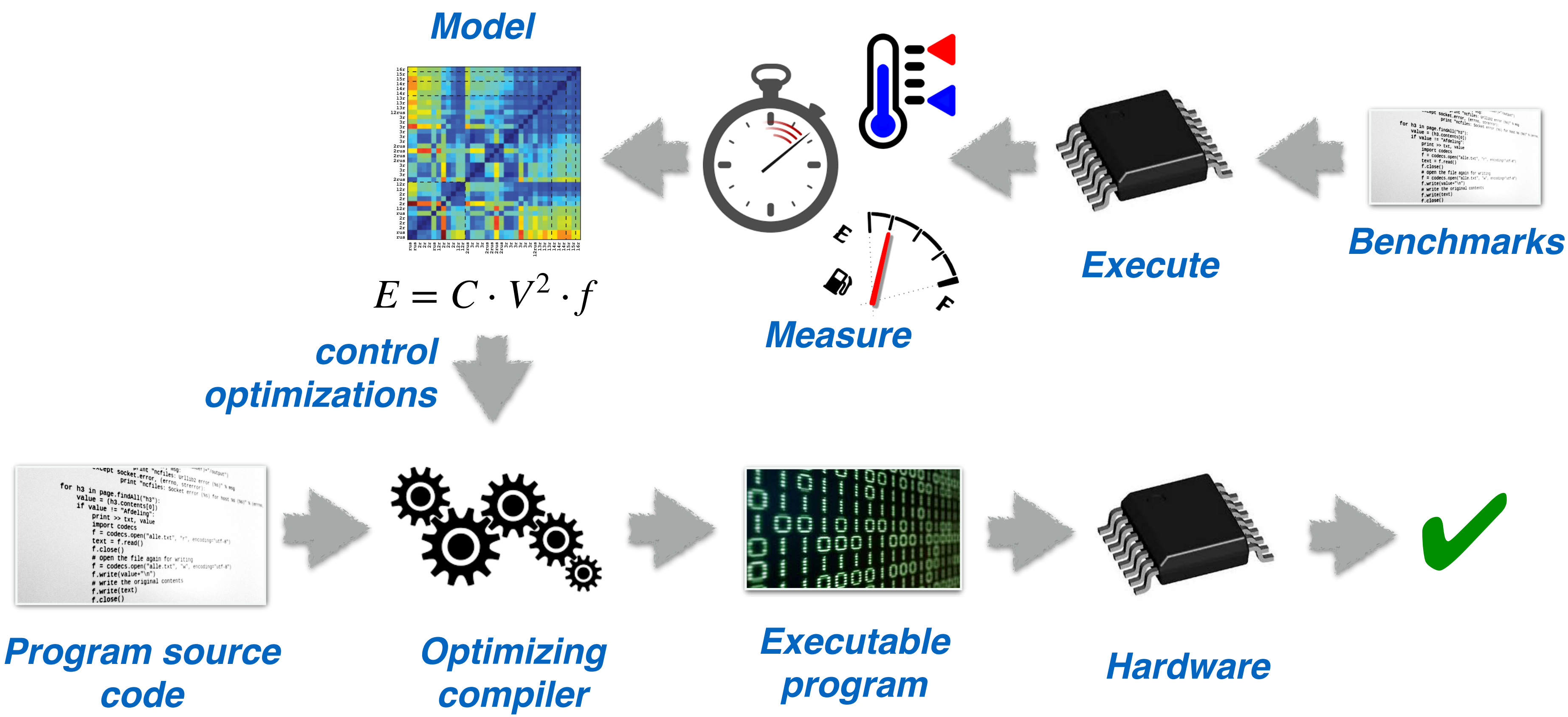
Optimization approaches

- Compile, run, measure, change options and repeat… [2,3]



Optimization approaches

- Integrate models of non-functional parameters into optimization decisions [4,5,6]



Example optimization: constant folding

Idea:

if operands are known at compile time, perform the operation **statically** (= once, during compilation)

```
int x = (2 + 3) * y → int x = 5 * y
b & false           → false
```

- What performance metric does it improve?
  - In general, the question whether an optimization improves performance is undecidable
- At which compilation step can it be applied?
  - Intermediate representation
  - After optimizations that create constant expressions

Example optimization: constant folding

```
int x = (2 + 3) * y → int x = 5 * y
```

- When is constant folding safely applicable?
  - for Boolean values: yes
  - for integer values: *almost always* yes
    - exception: division by zero
  - for floating point values: *caution*
    - e.g. rounding effects may lead to numerically different results
- General consideration of safety
  - Whether an optimization is safe depends on language semantics.
  - Languages that provide weaker guarantees to the programmer permit more optimizations, but have more ambiguity in their behavior – see e.g. [7]

Algebraic simplification

- More general form of constant folding
  - Makes use of mathematically sound simplification rules
- Identities:

```
a * 1     → a
a + 0     → a
b | false → b
```

- Associativity and commutativity rules:

```
(a + b) + c → a + (b + c)
a + b       → b + a
```

Algebraic simplification

- Combined with constant folding:

```
(a + 1) + 2 → a + (1 + 2) → a + 3
(2 + a) + 4 → (a + 2) + 4 → a + (2 + 4) → a + 6
```

- Iteration of these optimizations is useful – but how much?

Strength reduction

- Replace an expensive operation with a cheaper one:

```
a * 4     →  a << 2
a * 7     →  (a << 3) - a
a / 64    →  (a >> 6)
```

Division by non-power of 2 integer constants is more complex, see [8], Ch. 10-4

- Effectiveness of this optimization depends on the architecture
  - Useful if fast shifter (barrel shifter) is available

```
int foo(int a) {
  int z;
  z = a*7;
  return z;
}
```

clang -O0

```
imull   $7, -4(%rbp), %edi
```

clang -O3

```
leal  (,%rdi,8), %eax
subl  %edi, %eax
```

What’s next?

- Optimizations in detail: analyses and transformations

References

[1] Lars Wehmeyer, Manoj Kumar Jain, Stefan Steinke, Peter Marwedel and M. Balakrishnan.
Analysis of the Influence of Register File Size on Energy Consumption, Code Size and Execution Time.
IEEE TCAD 20-11, November 2001

[2] Pan, Zhelong & Eigenmann, R.. (2006).
Fast and effective orchestration of compiler optimizations for automatic performance tuning.
Proceedings of CGO 2006. DOI 10.1109/CGO.2006.38.

[3] Paul Lokuciejewski, Sascha Plazar, Heiko Falk, Peter Marwedel and Lothar Thiele.
Approximating Pareto optimal compiler optimization sequences-a trade-off between WCET, ACET and code size.
Software: Practice and Experience May 2011, DOI 10.1002/spe.1079

[4] Tiwari, V. and Malik, S. And Wolfe, A.
Power Analysis of Embedded Software: A FirstStep towards Software Power Minimization
IEEE, Trans. On VLSI Systems, December,1994

[5] Stefan Steinke, Markus Knauer, Lars Wehmeyer and Peter Marwedel.
An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations.
In PATMOS 2001, Yverdon (Switzerland), September 2001

[6] Neville Grech et al., 2015
Static analysis of energy consumption for LLVM IR programs
In Proceedings SCOPES ’15. ACM. DOI:https://doi.org/10.1145/2764967.2764974

[7] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013.
Towards optimization-safe systems: analyzing the impact of undefined behavior.
In Proceedings of SOSP ’13. ACM. DOI:https://doi.org/10.1145/2517349.2522728

[8] Henry S. Warren, Jr. Hacker’s Delight, 2nd Edition, Addison-Wesley 2012, ISBN 978-0-321-84268-8