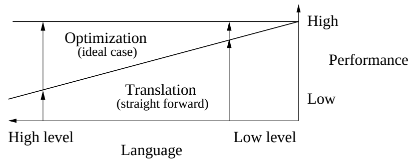
Optimizing Compilers

Introduction

Markus Schordan

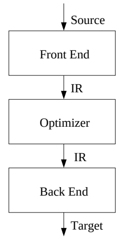
Institut für Computersprachen
Technische Universität Wien

Languages and Performance



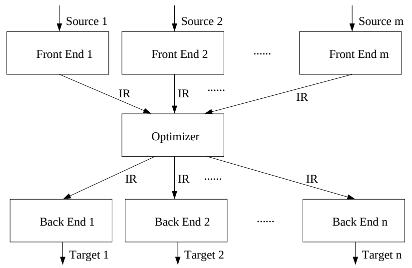
- Common perception that high level languages/abstraction gives low level of performance.
- Translation (straight forward) preserves semantics but does not exploit specific opportunities of lower level language with respect to performance.
- Optimization improves performance (misnomer: usually we do not achieve an "optimal" solution - but it is the ideal case)

Optimizing Compiler



Goal of code optimization. Discover, at compile-time, information about the run-time behavior of the program and use that information to improve the code generated by the compiler.

Optimizing Compiler(s)



- Decouple Front End from Back End
  - without IR: m source languages, n targets $\rightarrow m \times n$ compilers
  - with IR: m Front Ends, n Back Ends
  - Problem: level of IR (possible solution: multiple levels of IR)

Intermediate Representation (IR)

- High level
  - quite close to source language
  - e.g. abstract syntax tree
  - code generation issues are quite clumsy at high-level
  - adequate for high-level optimizations (cache, loops)
- Medium level
  - represent source variables, temporaries, (and registers)
  - reduce control flow to conditional and unconditional branches
  - adequate to perform machine independent optimizations
- Low level
  - correspond to target-machine instructions
  - adequate to perform machine dependent optimizations

Different Kinds of Optimizations

- Speeding up execution of compiled code
- Size of compiled code
  - when committed to read-only memory where size is an economic constraint
  - or code transmitted over a limited-bandwidth communications channel
- Energy consumption
- Response to real-time events
- etc.

Considerations for Optimization

- Safety
  - correctness: generated code must have the same meaning as the input code
  - meaning: is the observable behavior of the program
- Profitability
  - improvement of code
  - trade offs between different kinds of optimizations
- Problems
  - reading past array bounds, pointer arithmetics, etc.

Scope of Optimization (1)

- Local
  - basic blocks
  - statements are executed sequentially
  - if any statement is executed the entire block is executed
  - limited to improvements that involve operations that all occur in the same block
- Intra-procedural (global)
  - entire procedure
  - procedure provides a natural boundary for both analysis and transformation
  - procedures are abstractions encapsulating and insulating run-time environments
  - opportunities for improvements that local optimizations do not have

Scope of Optimization (2)

- Inter-procedural (whole program)
  - entire program
  - exposes new opportunities but also new challenges
    - name-scoping
    - parameter binding
    - virtual methods
    - recursive methods (number of variables?)
  - scalability to program size

Optimization Taxonomy

Optimizations are categorized by the effect they have on the code.

- Machine independent
  - largely ignore the details of the target machine
  - in many cases profitability of a transformation depends on detailed machine-dependent issues, but those are ignored
- Machine dependent
  - explicitly consider details of the target machine
  - many of these transformations fall into the realm of code generation
  - some are within the scope of the optimizer (some cache optimizations, some expose instruction level parallelism)

Machine Independent Optimizations (1)

- Dead code elimination
  - eliminate useless or unreachable code
  - algebraic identities
- Code motion
  - move operation to place where it executes less frequently
  - loop invariant code motion, hoisting, constant propagation
- Specialize
  - to specific context in which an operation will execute
  - operator strength reduction, constant propagation, peephole optimization

Machine Independent Optimizations (2)

- Eliminate redundancy
  - replace redundant computation with a reference to previously computed value
  - e.g. common subexpression elimination, value numbering
- Enable other transformations
  - rearrange code to expose more opportunities for other transformations
  - e.g. inlining, cloning

Machine Dependent Optimizations

- Take advantage of special hardware features
  - Instruction Selection
- Manage or hide latency
  - Arrange final code in a way that hides the latency of some operations
  - Instruction Scheduling
- Manage bounded machine resources
  - Registers, functional units, cache memory, main memory

Example: C++STL Code Optimization

- Different programming styles for iterating on a container and performing operation on each element
- Use different levels of abstractions for iteration, container, and operation on elements
- Optimization levels O1-3 compared with GNU 4.0 compiler

Concrete example: We iterate on container 'mycontainer' and perform an operation on each element

- Container is a `vector`
- Elements are of type `numeric_type` (`double`)
- Operation of adding 1 is applied to each element
- Evaluation Cases EC 1-6

Acknowledgement: Joint work with Rene Heinzl

Programming Styles - 1,2

EC1: Imperative Programming

```
for (unsigned int i = 0; i < mycontainer.size(); ++i) {
    mycontainer(i) += 1.0;
}
```

EC2: Weakly Generic Programming

```
for (vector<numeric_type>::iterator it = mycontainer.begin();
   it != mycontainer.end();
   ++it) {
     *it += 1.0;
}
```


Programming Style - 3

EC3: Generic Programming

```
for_each(mycontainer.begin(),
         mycontainer.end(),
         plus_n<numeric_type>(1.0) );
```

Functor

```
template<class datatype>
struct plus_n
{
  plus_n(datatype member):member(member) {}
  void operator()(datatype& value) { value += member; }
private:
  datatype member;
};
```


Programming Style - 4

EC4: Functional Programming with STL

```
transform(mycontainer.begin(),
          mycontainer.end(),
          mycontainer.begin(),
          bind2nd(std::plus<numeric_type>(), 1.0));
```

- plus: binary function object that returns the result of adding its first and second arguments
- bind2nd: Templatized utility for binding values to function objects

Programming Styles - 5,6

Functional Programming with Boost::lambda

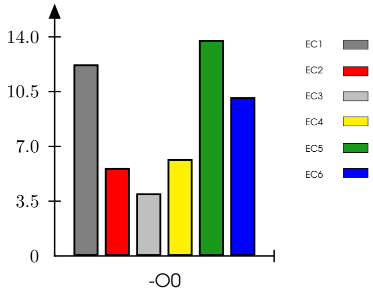
```
std::for_each(mycontainer.begin(),
              mycontainer.end(),
              boost::lambda::_1 +=1.0 );
```

Functional Programming with Boost::phoenix

```
std::for_each(mycontainer.begin(),
              mycontainer.end(),
              phoenix::arg1 += 1.0 );
```

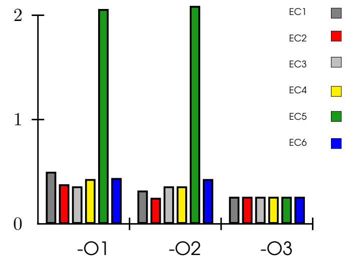
- Use of unnamed function object.

Evaluation (EC1-6 without optimization)



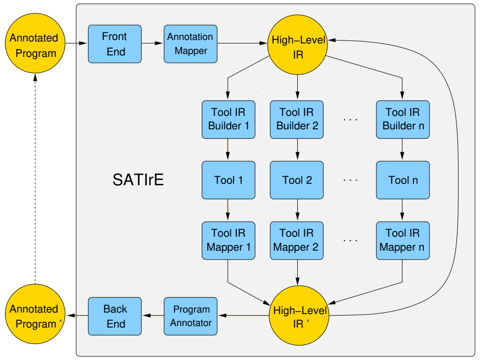
- Compiler: GNU g++ 4.0
- Evaluation Cases 1-6
- Time measured in milliseconds, container size: 1000

Evaluation: Optimization Levels O1-3

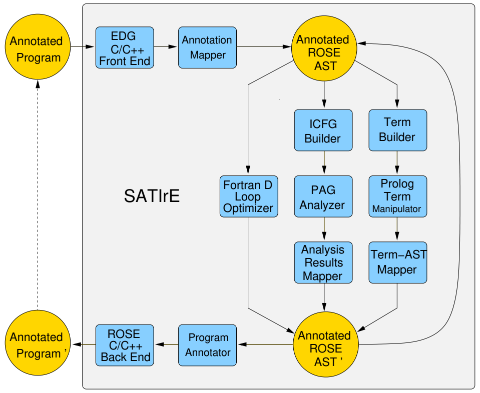


- Compiler: GNU g++ 4.0
- The actual run-time with different optimization levels `-O1`, `-O2`, `-O3` for each programming style (EC 1-6)
- An almost identical run-time is achieved at level `-O3`.

SATIrE: Static Analysis and Tool Integration Engine



SATIrE: Concrete Architecture (Oct'07)



SATIrE: Components (1)

- C/C++ Front End (Edison Design Group)
- Annotation Mapper (maps source-code annotations to an accessible representation in the ROSE-AST)
- Program Annotator (annotates programs with analysis results; combined with the Annotation Mapper this allows to make analysis results persistent in source-code for subsequent analysis and optimization)
- C/C++ Back End (generates C++ code from ROSE-AST)

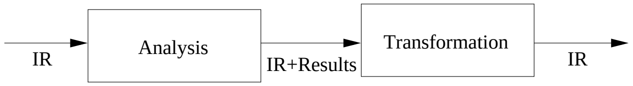
SATIrE Components (2)

- Integration 1 (Loop Optimizer)
  - Loop Optimizer: ported from the Fortran D compiler and integrated in LLNL-ROSE
- Integration 2 (PAG)
  - ICFG Builder: Interprocedural Control Flow Graph Generator, addresses full C++
  - PAG Analyzer: a program analyzer, generated with AbsInt's Program Analysis Generator (PAG) from a user-specified program analysis
  - Analysis Results Mapper: Maps Analysis Results from ICFG back to ROSE-AST, makes them available as AST-Attributes

SATIrE Components (3)

- Integration 3 (Termite)
  - Term Builder: generates an external textual term representation of the ROSE-AST (Term is in Prolog syntax)
  - Term-AST Mapper: parses the external textual program representation and translates it into a ROSE-AST

Optimization



- Analysis
  - determine properties of program
  - safe, pessimistic assumptions
- Transformation
  - based on analysis results

The Essence of Program Analysis

Program analysis offers techniques for predicting statically at compile-time safe and efficient *approximations* to the set of configurations or behaviors arising dynamically at run-time.

Safe : faithful to the semantics

Efficient : implementation with

- good time performance
- low space consumption

Typical Optimization Aspects

- Avoid redundant computations
  - reuse available results
  - move loop invariant computations outside loops
- Avoid superfluous computations
  - results known not to be needed
  - results known already at compile time

to be demonstrated in some examples ...

Lowering / IR / Address Computation

```
int a[m][n], b[m][n], c[m][n];
...
for(int i=0; i<m; ++i) {
  for(int j=0; j<n; ++j) {
    a[i][j]=b[i][j]+c[i][j];
  }
}
```

```
i=0;
while(i<m) {
  j=0;
  while(j<n) {
    temp=Base(a)+i*n+j;
    *(temp)=*(Base(b)+i*n+j)+*(Base(c)+i*n+j);
    j=j+1;
  }
  i=i+1;
}
```

Remark: in C: `a[i] == Base(a)+i*n (size of element type implicit)`

Available Expressions Analysis

```
i=0;
while(i<m) {
  j=0;
  while(j<n) {
    temp = (Base(a)+i*n+j);
    *temp = *(Base(b)+ i*n+j ) + *(Base(c)+ i*n+j );
    j=j+1;
  }
  i=i+1;
}
```

- Determines for each program point, which expression *must* have already been computed, and not later modified, on all paths to the program point.

Common Subexpression Elimination

```
i=0;
while(i<m) {
  j=0;
  while(j<n) {

    temp = (Base(a)+ i*n+j );
    *temp = *(Base(b)+ i*n+j )
        + *(Base(c)+ i*n+j );
    j=j+1;
  }
  i=i+1;
}
```

```
i=0;
while(i<m) {
  j=0;
  while(j<n) {
    t1=i*n+j ;
    temp = (Base(a)+ t1 );
    *temp = *(Base(b)+ t1 )
        + *(Base(c)+ t1 );
    j=j+1;
  }
  i=i+1;
}
```

- Analysis: Available Expressions Analysis
- Transformation: Eliminate recomputations of `i*n+j`
  - Introduce `t1=i*n+j`
  - Use `t1` instead of `i*n+j`

Detection of Loop Invariants

```
i=0;
while(i<m) {
  j=0;
  while(j<n) {
    t1= i*n +j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1) + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
}
```

Loop Invariant: Expression that is always computed to the same value each iteration of the loop.

Loop Invariant Code Motion

```
i=0;
while(i<m) {
  j=0;

  while(j<n) {
    t1= i*n +j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
        + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
}
```

```
i=0;
while(i<m) {
  j=0;
  t2=i*n ;
  while(j<n) {
    t1= t2 +j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
        + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
}
```

- Analysis: loop invariant detection
- Transformation: move loop invariant outside loop
  - introduce `t2=i*n` and replace `i*n` by `t2`
  - move `t2=i*n` outside loop

Detection of Induction Variables

```
i=0 ;
while(i<m) {
  j=0;
  t2=i*n ;
  while(j<n) {
    t1=t2+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
        + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1 ;
}
```

Basic Induction Variables. Variables i whose only definitions within a loop is of the form $i = i + c$ or $i = i - c$ and c is a loop invariant.

Derived Induction Variables. Variables j defined only once in a loop whose value is a linear function of some basic induction variable.

Strength Reduction (1)

A transformation that replaces a repeated series of expensive ("strong") operations with a series of inexpensive ("weak") operations that compute the same values.

Classic example: replaces integer multiplications based on a loop index with equivalent additions.

- This particular case arises routinely from expansion of array and structure addresses in loops.

Strength Reduction (2)

```
i=0 ;

while(i<m) {
  j=0;
  t2=i*n ;
  while(j<n) {
    t1=t2+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
        + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1 ;

}
```

```
i=0;
t3=0 ;
while(i<m) {
  j=0;
  t2= t3 ;
  while(j<n) {
    t1=t2+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
        + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n ;
}
```

The multiplication `i*n` is replaced by successive additions.

Copy Analysis

```
i=0;
t3=0;
while(i<m) {
  j=0;
  t2=t3;
  while(j<n) {
    t1=t2+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
          + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n;
}
```

Determines for each program point, which copy statements $x = y$ that still are relevant (i.e. neither x nor y have been redefined) when control reaches that point.

Copy Propagation

```
i=0;
t3=0;
while(i<m) {
  j=0;
  t2=t3;
  while(j<n) {
    t1=t2+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
          + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n;
}
```

```
i=0;
t3=0;
while(i<m) {
  j=0;
  t2=t3;
  while(j<n) {
    t1=t3+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
          + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n;
}
```

- Analysis: Copy Analysis and def-use chains (ensure only one definition reaches the use of x)
- Transformation: Replace the use of x by y.

Live Variables Analysis

- A variable is *live* at a program point if there is a path from this program point to a use of the variable that does not re-define the variable.
- Determines for each program point, which variable *may* be live at the exit from that point.
- If a variable is not live, it is dead.

Live Variables Analysis

```
i=0;
t3=0;
while(i<m) {
  j=0;
  t2=t3;
  while(j<n) {
    t1=t3+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
          + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n;
}
```

- Only dead variables are marked.

Dead Code Elimination

```
i=0;
t3=0;
while(i<m) {
  j=0;
  t2=t3;
  while(j<n) {
    t1=t3+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
          + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n;
}
```

```
i=0;
t3=0;
while(i<m) {
  j=0;

  while(j<n) {
    t1=t3+j;
    temp = (Base(a)+t1);
    *temp = *(Base(b)+t1)
          + *(Base(c)+t1);
    j=j+1;
  }
  i=i+1;
  t3=t3+n;
}
```


Example: Optimizations Summary

Analysis	Transformation
Available expr. analysis	Common subexpr. elim.
Loop invariant detection	Invariant code motion
Induction variable detection	Strength reduction
Copy analysis	Copy propagation
Live variables analysis	Dead code elimination

Further optimizations?

Pointer/Alias/Shape Analysis

- Ambiguous memory references interfere with an optimizer's ability to improve code.
- One major source of ambiguity is the use of pointer-based values.

Goal: determine for each pointer the set of memory locations to which it may refer.

Without such analysis the compiler must assume that each pointer can refer to any addressable value, including

- any space allocated in the run-time heap
- any variable whose address is explicitly taken
- any variable passed as a call-by-reference parameter

Forms of pointer analysis: points-to sets, alias pairs, shape analysis.

Optimizations for Object-Oriented Languages (1)

Invoking a method in an object-oriented language requires looking up the address of the block of code which implements that method and passing control to it.

Opportunities for optimization

- Look-up may be performed at compile time
- Only one implementation of the method in the class and in its subclasses
- Language provides a declaration which forces the call to be non-virtual
- Compiler performs static analysis which can determine that a unique implementation is always called at a particular call-site.

Optimizations for Object-Oriented Languages (2)

Optimizations:

- Dispatch Table Compression
- Devirtualization
- Inlining
- Escape Analysis for allocating objects on the run-time stack (instead of heap)

References (1)

- Material for this 1st lecture

  `www.complang.tuwien.ac.at/markus/optub.html`
- Book

  Steven S. Muchnick:

  Advanced Compiler Design and Implementation, Morgan Kaufmann; (856 pages, ISBN: 1558603204), 1997.

  Chapter 1 (Introduction)
- Book

  Keith D. Cooper, Linda Torczon:

  Engineering a Compiler, Morgan Kaufmann; (801 pages, ISBN: 155860698X), 2003.

  Chapter 1 (Introduction), 10 (Scalar Optimizations)

References (2)

- Book

  Flemming Nielson, Hanne Riis Nielson, Chris Hankin:

  Principles of Program Analysis.

  Springer, (2nd edition, 452 pages, ISBN 3-540-65410-0), 2005.
  - Chapter 1 (Introduction)