Advanced C Programming
Compilers II

Sebastian Hack
hack@cs.uni-sb.de

Christoph Weidenbach
weidenbach@mpi-inf.mpg.de

Winter Term 2008/09
Today: A small high-level glance at some compiler optimizations

Data-Dependence Graphs

Optimizations on the IR
- Constant Folding
- Common Subexpression Elimination
- Operator Strength Reduction in Loops

Backend Optimizations
- Instruction Selection
- Instruction Scheduling
- Register Allocation
Data-Dependence Graphs

- In SSA, every variable has only one definition in the code
- The instruction that defines the variable and the variable can be identified
- We do not need variables anymore
- SSA removes output dependences
- Represent instructions in a data-dependence graphs
- If an instruction has multiple return values (i.e. `divmod`) use tuples and projection instructions
- Inside a basic block, graph is acyclic
Data-Dependence Graphs

\[ T \leftarrow \text{start} \]
\[ M \leftarrow \text{proj}(T, \text{mem}) \]
\[ A \leftarrow \text{proj}(T, \text{args}) \]
\[ a_1 \leftarrow \text{proj}(A, 0) \]
\[ a_2 \leftarrow \text{proj}(A, 1) \]
\[ a_3 \leftarrow \text{add}(a_1, a_2) \]
\[ \text{return}(M, a_3) \]
Simple Scalar Transformations
Constant Folding, Strength Reduction, Algebraic Identities

- All constant expressions are evaluated
- On SSA graphs these are just graph transformations
- When cross-compiling: Obey target machine arithmetic!
- Be careful when operations cause side-effects:

```c
int main () {
    int x = 5 / 0;
    return 0;
}
```

trap must also be caused when program is run

- Optimize all algebraic identities
  \(x + 0, x \& 0, x \cdot 1, x - x, \ldots\)

- Reduce strength of operators
  \(2 \cdot x \rightarrow x + x, 5 \cdot x \rightarrow x \ll 2 + x, \text{ and so on}\)

- They come not only from user code but are left over by other optimizations
Simple Scalar Transformations
Constant Folding, Strength Reduction, Algebraic Identities

- Normalize expressions for commutative operations

- Interplay of several small local optimizations

\[
(1 - x) + 2 \\
= (1 + (-x)) + 2 \quad \text{Normalize \ - \ to \ +} \\
= -x + (1 + 2) \quad \text{Associativity} \\
= -x + 3 \quad \text{Fold constant} \\
= 3 - x \quad \text{Local optimize}
\]
Common Subexpression Elimination (CSE)

- Goal: Avoid recomputation of equal expressions
- Again:
  - Not only in code written explicitly by the programmer
  - Also stems from address arithmetic, other optimizations
- Advantages:
  - Save computations
- Disadvantages:
  - Possibly increases register pressure
  - Constants often do not have to be materialized in a register
Common Subexpression Elimination (CSE)

Example

- Address arithmetic of an access of a struct in an array

```c
struct pt {
    int x, y;
};

int foo(struct pt *arr) {
    int i;
    ...
    arr[i].x = ...;
    arr[i].y = ...;
}
```

The frontend produces:

- $p \leftarrow \text{param}(0)$
- $a_1 \leftarrow \text{mul}(i, 8)$
- $a_2 \leftarrow \text{add}(a_1, p)$
- $a_3 \leftarrow \text{add}(a_2, 0)$
- $M_2 \leftarrow \text{store}(M_1, a_3, v_1)$
- $a_4 \leftarrow \text{mul}(i, 8)$
- $a_5 \leftarrow \text{add}(a_4, p)$
- $a_6 \leftarrow \text{add}(a_5, 4)$
- $M_3 \leftarrow \text{store}(M_2, a_6, v_2)$

- $a_2$ and $a_5$ have always the same value
- The common subexpressions can be eliminated
Common Subexpression Elimination (CSE)

Example

- Address arithmetic of an access of a struct in an array

```c
struct pt {
    int x, y;
};

int foo(struct pt *arr) {
    int i;
    ...
    arr[i].x = ...;
    arr[i].y = ...;
}
```

Optimized version

```c
p ← param(0)
a₁ ← mul(i, 8)
a₂ ← add(a₁, p)
M₂ ← store(M₁, a₂, v₁)
a₆ ← add(a₂, 4)
M₃ ← store(M₂, a₆, v₂)
```

- \( a₂ \) and \( a₅ \) have always the same value
- The common subexpressions can be eliminated
Common Subexpression Elimination

How does it work?

▶ The simple version is restricted to a basic block
▶ Can easily be extended to dominators
▶ We can compare two instruction (variables) for equality easily:
  ▶ Operator the same
  ▶ Operands pairwise equal (recursive check)
▶ Maintain a hash table for every basic block that holds all instructions of the block
▶ Hash code of an instruction derived from hash code of the operands and from the operator
▶ Whenever we want to add an instruction to the block, look in the hash table whether the expression is already computed
▶ Without SSA, that would not be that simple!
  ▶ Multiple definitions of the same variable possible!
▶ Again:
  ▶ Everything only for scalar, alias-free variables
  ▶ Cannot look inside the memory
Common Subexpression Elimination

Again those potentially aliased variables...

```c
int foo(int i, int *p) {
    int x = *p + i;
    int y = x * 2;
    ...
    int a = *p + i;
    int y = x * 2;
}
```

- Depending on the code in the middle it may be hard to do CSE
- Compiler might not be able to prove that there no aliased access to *p

```c
int foo(int i, int *p) {
    int dp = *p;
    int x = dp + i;
    int y = x * 2;
    ...
    int a = dp + i;
    int y = x * 2;
}
```

- User knows p is alias free
- CSE can be done on expressions at the end
Common Subexpression Elimination

... and register pressure

- Consider following example again

\[
\begin{align*}
p & \leftarrow \text{param}(0) \\
a_1 & \leftarrow \text{mul}(i, 4) \\
a_2 & \leftarrow \text{add}(a_1, p) \\
\ell_1 : M_2 & \leftarrow \text{store}(M_1, a_2, v_1) \\
\vdots \\
\ell_2 : a_6 & \leftarrow \text{add}(a_2, 4) \\
M_3 & \leftarrow \text{store}(M_2, a_6, v_2) \\
\vdots \\
& \leftarrow \tau(a_1, \ldots)
\end{align*}
\]

- Between \(\ell_1\) and \(\ell_2\) both, \(a_1\) and \(a_2\) are live
- Two registers would be occupied with \(a_1\) and \(a_2\)
- If the register pressure is very high between \(\ell_1\) and \(\ell_2\) one of both might be spilled
- Perhaps recomputing \(\text{add}(a_1, p)\) would be better
- Could have inserted loads and stores to save an addition(!)
Definition (Liveness)

A variable $v$ is live at an instruction $\ell$ if there is a path from $\ell$ to a use of $v$ that does not go through the definition.

Definition (Register Pressure)

The number of simultaneously live variables at an instruction $\ell$ is called the register pressure at $\ell$.

- CSE might increase register pressure
- Depends on the register file size of the machine
- IR is unaware of the constraints of the machine
Variables that are linearly dependent on the loop counter

```c
for (i = 0; i < n; i++) {
    int j = 25 * i;
    ...
}
```

Multiplication in the loop is potentially expensive.

Compiler rewrites it to:

```c
for (i = 0, j = 0; i < n; i++, j += 25) {
    ...
}
```

However, we now have two variables live in the loop.

- Kills multiplications, but raises register pressure
  - careful trade-off needed!
Example

Why is that useful? Array addressing:

```c
for (i = 0; i < n; i++) {
    a[i] = 2 * b[i];
}
```
Operator Strength Reduction in Loops (OSR)

Example

Why is that useful? Array addressing:

```c
for (i = 0; i < n; i++) {
    a[i] = 2 * b[i];
}
```

really is:

```c
for (i = 0; i < n; i++) {
    *(a + sizeof(*a) * i) = 2 * *(b + sizeof(*b) * i);
}
```
Operator Strength Reduction in Loops (OSR)

Example

- Why is that useful? Array addressing:

```c
for (i = 0; i < n; i++) {
    a[i] = 2 * b[i];
}
```

- really is:

```c
for (i = 0; i < n; i++) {
    *(a + sizeof(*a) * i) = 2 * *(b + sizeof(*b) * i);
}
```

- can be rewritten to:

```c
pa = a; pb = b;
for (i = 0; i < n; i++) {
    *pa = 2 * *pb;
    pa += sizeof(*a); pb += sizeof(*b);
}
```
Operator Strength Reduction in Loops (OSR)

Example

Why is that useful? Array addressing:

```c
for (i = 0; i < n; i++) {
    a[i] = 2 * b[i];
}
```

really is:

```c
for (i = 0; i < n; i++) {
    *(a + sizeof(*a) * i) = 2 * *(b + sizeof(*b) * i);
}
```

can be rewritten to:

```c
pa = a; pb = b;
for (i = 0; i < n; i++) {
    *pa = 2 * *pb;
    pa += sizeof(*a); pb += sizeof(*b);
}
```

When we do not need the loop counter at all:

```c
pa = a; pb = b; m = a + sizeof(*a) * n;
for (; a < m; ) {
    *pa = 2 * *pb;
    pa += sizeof(*a); pb += sizeof(*b);
}
```
Never do this yourself

Confer to alias problems from last lecture:
a[i] is better analyzable than *a++

The compiler can do it easily for all variables (scalar, alias-free!) that are linearly dependent on the loop counter
Inlining

Remove function calls by pasting-in the body of called function at the call site

Advantages:
- Save overhead for call:
  - Saving the return address, the call
  - Moving parameters to specific registers or on the stack: memory operations
- Function body can be optimized within context of caller
- If the body is small, call overhead might be larger than executed code of the body

Disadvantages:
- Potential code bloat
- Larger instruction cache footprint

Limitations:
- Indirect calls hard to inline: need to know where it goes
- Especially severe in OO-programs (dynamic dispatch)
Inlining

Example

- Scalar Product of a 2D point encapsulated in a function
- foo just forms the required struct and copies arguments in it
- These copies are just there to satisfy the signature of sprod

```c
float sprod(struct pt *p) {
    return p->x * p->x + p->y * p->y;
}

float foo(float x, float y) {
    struct pt p;
    p.x = x;
    p.y = y;
    return sprod(&p);
}
```

- After inlining the body of sprod

```c
float foo(float x, float y) {
    struct pt p;
    p.x = x;
    p.y = y;
    return p.x * p.x + p.y * p.y;
}
```

- cont’d
Inlining

Example

- p is still kept in memory (on the call stack of \texttt{foo})
- p.x = \ldots \ results in memory stores and \ldots = p.x in loads
- To remove these stores and loads the compiler has to prove that there are no aliased accesses inbetween
- Easy in this case
- After load/store optimizations and some scalarization

\begin{verbatim}
float foo(float x, float y) {
    float t1 = x;
    float t2 = y;
    return t1 * t1 + t2 * t2;
}
\end{verbatim}

- And finally copy propagation

\begin{verbatim}
float foo(float x, float y) {
    return x * x + y * y;
}
\end{verbatim}

- We get what we want
Inlining

Summary

- Indispensable for small functions (getters, setter, ...)
- Allows to implement abstraction with functions efficiently
- Beware of function pointers!
- Polymorphism in OO languages are function pointers hidden under a nice syntax!

- Small functions like `sprod` should go in header files to be inlineable:

  ```c
  static inline float sprod(const struct pt *p) {
    return p->x * p->x + p->y * p->y;
  }
  ```

- If you put them in the `.c` file you need whole-program compilation
- Cannot compile every `.c` separately
  ... or inlining has to be done by the linker 😊
Contents

Data-Dependence Graphs

Optimizations on the IR
  Constant Folding
  Common Subexpression Elimination
  Operator Strength Reduction in Loops

Backend Optimizations
  Instruction Selection
  Instruction Scheduling
  Register Allocation
Overview

- Implement the constraints of the target processor
- Some machines are harder, some easier
- Some have very wild constraints that are hard to tackle algorithmically
- Hardware designers thought to do something very smart...
- ...compiler writers are just sighing
- The hardware guys should have to write the code generator!

- Some examples:
  - On Sparc, doubles start at even register numbers:
    Turns optimal RA in basic block NP-complete
  - Split register files for address generation and normal computations on some DSPs: Have to minimize moves between register files
  - Parts of a register are accessible under a different name
  - and many more...

- All these render the backend’s task often NP-complete on straight-line code
- The best is: A standard RISC machine like Alpha
Principal Phases in a Backend

Back End

Instruction Selection
Select processor instructions for operations in the IR

Instruction Scheduling
Create a linear order of the instructions

Register Allocation
Map variables of the IR to processor’s registers

Bad news
All three phases are NP-complete and inter-dependent
Instruction Selection

- IR operator set is as minimalistic as possible
- Processors often have more instructions than the operators of the IR
- Interdependences with register allocation:
- Interdependences with scheduling:
  - Not every instruction can be decoded by every decoder
  - Not every instruction can be executed by every functional unit
  - Latency can depend on instructions before/after
  - Not to talk about things like \( \mu \)-op fusion and so on
Instruction Selection

- x86 has the powerful `lea` instruction that computes

\[
\text{lea } r1, [r2 + r3 \times \text{scale}] + \text{imm} \iff r1 \leftarrow r2 + r3 \cdot \text{scale} + \text{imm}
\]

for \( \text{scale} \in 1, 2, 4, 8 \) and \( 0 \leq \text{imm} \leq 2^{32} - 1 \) using the addressing path

- Many CPUs feature a multiply-and-add instruction

\[
r1 \leftarrow r2 \cdot r3 + r4
\]

because it is easy to implement in hardware and occurs often in practice

- Digital signal processors (DSPs) often have more complex instructions to support fixed-point arithmetic and operations common in video-/audio codecs

- Post-increment loads/stores on ARM/PowerPC
Instruction Scheduling

- Order the instructions linearly such that instruction level parallelism can be exploited by the CPU
- Not that important for out-of-order CPUs
- Recent Intel CPUs are in-order again!
- There scheduling is important since the processors fills the pipelines depending on the order in the instruction stream
- VLIW processors allow the compiler to fill the pipelines directly
- There scheduling is very important
- Instruction-level parallelism increases register pressure
- Strong interdependence with register allocation
Register Allocation

- Put as many variables in registers as possible
- Access to registers at least $3 \times$ faster than cache access
- Good register allocation is decisive for program performance
- What to do if there are more scalar, alias-free variables alive than registers?
- Some variables have to be spilled to memory
- Assume that instructions are already linearly ordered
  - Necessary because we need to know where an instruction is live
- Interdependences to instruction selection:
  - Inserts new instructions (spill code)
  - Could also rematerialize (recompute)
- Interdependences to scheduling:
  - Register pressure dominated by scheduling
  - Amount of inserted spill code determined by schedule
  - Need to reschedule after spill-code insertion because instruction stream changed
Register Allocation

- Clever heuristics exist
- Live ranges of variables are split around high-pressure areas where they are not used
- Provide as many scalar, alias-free variables as possible
- The compiler can then figure out when to put which in memory
- Much easier for the compiler than the other way around!