Outline

- Introduction to SSA
- Construction, Destruction
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

Analyses and Optimizations

- Analyses in Software Tech. support programmers
- Analyses in Compiler Construction allow to safely perform optimizations
- Cost model: runtime of a program
  - Statically only conservative approximations
    - Loop iterations
    - Conditional code
  - Even for linear code not known in advance:
    - Instruction scheduling
    - Cache access is data dependent
    - Instruction pipelining: execution time is not the sum of individual operations costs
- Alternative cost models:
  - memory size, power consumptions
  - Same non-decidability problem as for execution time
- Caution: cost of a program ≠ sum of costs of its elements

Optimization: Implementation

- Legal transformations in SSA-Graphs:
  - Simplifying transformations reduce the costs of a program
  - Preparative transformations allow the application of simplifying transformations
- Using
  - Algebraic Identities (e.g. Associative / Distributive law for certain operations)
  - Moving of operations
  - Reduction of dependencies
- Optimization is a sequence of goal directed, legal simplifying and legal preparative transformations
- Legibility proven
  - Locally by checking preconditions
  - Due to static data-flow analyses

Algebraic Identity: Elimination of Operations and its Inverse

Graph Rewrite Schema

Elimination of Memory Operation and its Inverse

SSA-subgraph before Transformation

SSA-subgraph after Transformation

No side effects in

Precondition established by local check preparative analyses

No side effects in

Load

Store

Load

Store

[a] = void
Elimination of Duplicated Memory Operations

Elimination of non-essential dependencies

Algebraic Identity: Invariant Compares

Associative Law

Distributive Law

Operator Simplification
### Constant Folding

#### Evaluation using source algebra or target algebra (if allowed by source language)

- **Constant Folding**
- \( \tau \) \( x \) \( y \) \( \tau \) \( x \ y \)

### Constant folding over \( \phi \)-functions

- **General: Moving arithmetic operations over \( \phi \)-functions**
- No side effects in \( \tau \) (no call, store)

### Optimizations

- **Strength reduction:**
  - Bauer & Samelson 1959
  - Replace expensive by cheap operations
  - Loops contain multiplications with iteration variable, these operations could be replaced by add operations (Induction analysis)
  - One of the oldest optimizations: already in Fortran I-compiler (1954/55) and Algol 58/60-compiler

- **Partial redundancy elimination (PRE):**
  - Morel & Renvoise 1978
  - Eliminate partially redundant computations
  - SSA eliminates all static but not dynamic redundancies
  - Problem on SSA: which is the best block to perform the computation
  - Move loop invariant computations out of loops, into conditionals
  - Subsumes a number of simpler optimization

### Example: Strength reduction

For \( i = 0; i < n; i++ \) {
  for \( j = 0; j < n; j++ \) {
    \( a[i,j] = a[i,j] \)
  }
}

// Original loop body:
\( a_0 = i*n*d + j*d \)
\( a_{ij} = a[i][j] + a_0 \)
\( b_0 = i*n*d + j*d \)
\( b_{ij} = b[i][j] + b_0 \)
\( c_{ij} = c[i][j] + c_0 \)

adda = a[0,0] + c_0
addb = b[0,0] + c_0
d = 4
addend = adda + n*n*d;
LOOP: adda = adda + adda;
      addb = addb + addb;
      jump LOOP
END: exit

### Induction Analysis Idea

- **Find induction variable** \( i \) for a loop using DFA
- \( i \) is induction variable if in loop only assignments of form
  \( i := i + c \) with loop constant \( c \) or, recursively,
  \( i := c' * i + c'' \) with \( i' \) induction variable and loop constants \( c', c'' \)
- \( c \) is a loop constant if \( c \) does not change value in loop, i.e.
  - \( c \) is static constant.
  - \( c \) computed in enclosing loop

- **Example (cont’d), consider the inner loop:**
  for \( (j = 0; j < n; j++) \) {
    \( \text{Direct induction variable: } j \), implicit \( j = j + 1 \) (\( c = 1 \))
    \( \text{Indirect induction variable: } a_0 = i*n*d + d \) \( (c = n, c'' = n*d) \)
    \( \text{Note that } i*n*d \text{ and } d \text{ a loop constants for the inner loop} \)
Induction Transformation Idea

- Transformation goal: values of induction variables should grow linearly with iteration; add operations replace multiplications

- Transformation:
  - Let $i_0$ initialization of $i$ and induction variables, $i := i + c$ and $i' := c + c''$
  - New variable $i_a$ initialized: $i_a := c \times i_0 + c''$
  - At loop end insert $i_a := i_a + c'$
  - Replace consistently operands $i'$ by $i_a$
  - Remove all assignments to $i$, $i'$ and $i_a$ themselves if $i$ is not used elsewhere (DFA)

Example:
- Before: \( \text{loop } ao = i \times n \times d + j \times d \ldots j++ \text{ end loop} \)
- After: \( \text{ao } = i \times n \times d \text{ loop } \ldots \text{ao } = \text{ao } + d \text{ end loop} \)

---

Induction Analysis: Implementation

- Assume initially optimistically: all variables are induction variables
- Finding induction variable $i$ for a loop follows definition
- Iteratively until fix point: $i$ is not induction variable if not:
  - $i := i_0 \times c$, with loop constants $c$ (direct induction variable)
  - $i := c \times i_0 \times c'$, with $i_0$ induction variable and loop constants $c, c'$ (indirect induction variable)
  - $i := \phi(i_1 \ldots i_n)$ with $i_0$ being direct induction variable
- On SSA, simplifications of that analysis are possible
  - any loop variable corresponds to a cyclic subgraph over a $\phi(i_1 \ldots i_n)$ node
  - Find Strongly Connected Component (SCC) and check those for the induction variable condition

---

Induction Variables

Direct Induction Variable Cycle

Induction Variables (Schematic)
Partial Redundancy Elimination: Idea

- SSA is representation
  - without (provable) static redundancies
  - with all dependencies explicit
- Question which block should contain the computation guaranteeing
  - that the result is used on all path to the end
  - that the computation is not repeatedly performed in loops
- First idea: compute each operation earliest (as soon as all arguments are available)
- Observation:
  - Fast introduction of many live values: high register pressure
  - Many execution path compute but do not use a certain value
- Solution is Partial Redundancy Elimination (PRE):
  - Delay computation until it is used on all paths
  - In practice: move them out of loops into conditional code
Observations on SSA

Operations that must be executed in original block:
1. \( \phi \)-nodes,
2. Computations with exceptions
3. Jumps
4. "pinned" operations (postponed)

1. Observation:
   All other nodes could be computed in other blocks as well if data dependencies are obeyed.

2. Observation:
   No statically redundant computation at all, i.e., one important goal of optimization immediately follows from the representation. Dynamic redundancy remains a problem.

Example: Initial Situation

Immature \( \phi' \)

Mature \( \phi' \rightarrow \phi \)

Placement of computations

Placing \( t_1 \) earliest
Placing $t_1$ latest

According of Knoop, Steffen

$\tau_1$ (re-)computed in each iteration.

Still $t_1$ (re-)computed in each iteration.

Insert Blocks

Virtually, insert empty blocks, to capture operations executed only on a specific path

Placing for $t_1$ out of loops into conditional code

However: Exist path $t_1$ not used on.

PRE: Discussion

- Placing earliest
  - Advantage: short code, could be fast code because of instruction cache; no unnecessary computations in loops; short jumps ...
  - Disadvantage: many paths do not need result, high register pressure
- Placing lazily
  - Advantage: computation needed on all paths
  - Disadvantage: unnecessary computations in loops
- Placing out of loops into conditional code
  - Advantage: no unnecessary computations in loops
  - Disadvantage: some unnecessary computations in general as some paths do not need result

PRE: Implementation

- Find partially (dynamically) redundant computations
  - If $\tau$ contains operation $t$ computing $\tau$, $\tau'$ contains $t$ consuming $\tau$
  - If $\tau'$ post-dominates $\tau$ ($\tau' \prec \tau$) no dynamic redundancy
  - Assume all operations as partially (dynamically) redundant
  - For each operation $\tau$ computing $t$ and the set of operations $\tau'_1, ..., \tau'_n$ consuming $\tau$, if $\beta(\tau'_k) \prec \beta(\tau)$ for some $k$ in $1, ..., n$ then $\tau$ is not placed partially redundant -- all others are (!)
- Eliminate partially redundant computations
  - Compute earliest position (all arguments available, all uses dominated) for each partially redundant operation $\tau$
  - Move copies of $\tau$ towards the consuming operations $\tau'_1, ..., \tau'_n$ along the dominator tree until no partial dynamic redundancies but stop at loop heads
  - Not deterministic but that does not matter (!)
**PRE: “Pinned” Operations**

- Placement sometimes only possible if it is the last transformation on SSA
- Same computation computed at several places
- Further optimizations recognize this wanted static redundancy

**Solution:**
- Let $t_1 : a_1 + b_1$ and $t_2 : a_1 + b_2$ semantic equivalent computations at different positions (blocks)
- Replace $+$ by a “pinned” $\oplus_{\text{Block}}$
- Thereby $\oplus$ operation additionally depends on the current block as new arguments
- Computations $a_1 \oplus b_1$ and $a_1 \oplus b_2$ not recognized as congruent any more

**Further Optimizations**

- Constant evaluation (simple transformation rule)
- Constant propagation (iterative application of that rule)
- Copy propagation (on SSA construction)
- Dead code elimination (on SSA construction)
- Common subexpression elimination (on SSA construction)
- Specialization of basic blocks, procedures, i.e. cloning
- Procedure inlining
- Control flow simplifications
- Loop transformations (Splitting/merging/unrolling)
- Bound check eliminations
- Cache optimizations (splitting, merging, unrolling)
- Parallelization
- …

**Observations**

- Order of optimizations matters in theory:
  - Application of one optimization might destroy precondition of another
  - Optimization can ruin the effects of the previous once
- Optimal order unclear (in scientific papers usual statements like: “Assume my optimization is the last …”)
- Simultaneous optimization too complex
- Usually first optimization gives 15% sum of remaining 5%, independent of the chosen optimizations
- Might differ in certain application domains, e.g. in numerical applications operator simplification gives factor >2, cache optimization factor 2-5

**Outline**

- Introduction to SSA
- Construction, Destruction
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

**Optimizations on Memory**

- Elimination of memory accesses.
- Elimination of object creations.
- Elimination non essential dependencies.
- Those are normalizing transformations for further analyses

**Nothing new under the sun:**

- Define abstract values, addresses, memory
- Define context-insensitive transfer functions for memory relevant SSA nodes (Load, Store, Call) (discussed already)
- Generalization to context-sensitive analyses (discussed already)
- Optimizations as graph transformations (discussed already)

**Memory Values**

- Differentiation by Name Schema (NS)
- Distinguish e.g.:
  - heap and stack
  - local arrays with different name
  - disjoint index sets in an array (odd/even etc.)
  - different types of heap objects
  - objects with same type but statically different creation program point
  - objects with same creation program point but with statically different path to that creation program point (execution context, context-sensitive)
Abstract values, addresses, memory

- References and values
  - allocation lattice $\mathbb{L}$ abstracts from values like integer or boolean
  - abstract heap memory $\mathcal{M}$
    - $O \xrightarrow{R} \mathbb{L}$ set of fields with reference object semantics
    - $O \xrightarrow{V} \mathbb{L}$ set of fields with value semantics
- Arrays
  - Treated as objects
  - abstract heap memory $\mathcal{M}$
    - $O \xrightarrow{R} \mathbb{L}$ set of fields with type array of reference object
    - $O \xrightarrow{V} \mathbb{L}$ set of fields with type array of value
- Abstract address $\text{Addr} \subseteq 2^F \times X$ ($F$ set of field names)
- object-field-(index) triples where index might be ignored

Updates of Memory

- Given an abstract object-field-(index) of a store operation
- In general, this abstract object points to more than one real memory cell
- A store operation overwrites only one of these cells, all others contain the same value
- Hence, a store to an abstract object-field-(index) adds a new possible (abstract) value - weak update
- Only if guaranteed that abstract object-field-(index) matches one and only one concrete address, a new (abstract) value overwrites the old value - strong update

Auxiliary function:

$$\text{update}(\mathcal{M}, a, \top, v) = \begin{cases} v & \text{if strong update possible} \\ \mathcal{M}(a, \top) \cup v & \text{otherwise, weak update} \end{cases}$$

Transfer functions (insensitive, no arrays)

- $T_{\text{store}}(\mathcal{M}, \text{Addr}, v) = (\mathcal{M}, \text{Addr} \cup \{\text{update}(\mathcal{M}, a, v)\})$
- $T_{\text{load}}(\mathcal{M}, \text{Addr}) = (\mathcal{M}, \text{Addr} \cup \{\mathcal{M}(a)\})$
- $T_{\text{alloc}}(\text{type}) = (\mathcal{M}, \{\text{alloc}(\text{type})\} \cup \{\text{update}(\mathcal{M}, o, n, \top) \rightarrow \bot\})$

Example: SSA

Initialization

No provably different memory addresses

Example: Main Loop Inner Product Algorithm

Initialization

Example:

Main Loop Inner Product Algorithm

No provably different memory addresses

Updates of Memory

Abstract values, addresses, memory
Actually two Iterators?

Memory objects replaced by values

Initialization: disjoint memory guaranteed

Value numbering proofs equivalence

Example revisited

Optimization only possible due to joint application of single techniques:
- Global analysis
- Elimination of polymorphism
- Elimination of non essential dependencies
- Elimination of memory operations
- Traditional optimizations