Outline

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

1

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

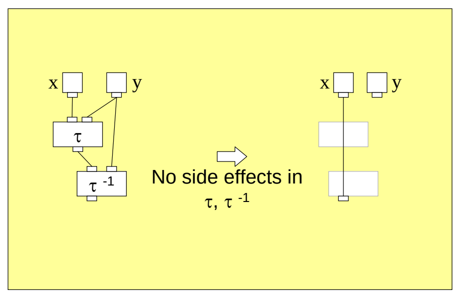
2

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

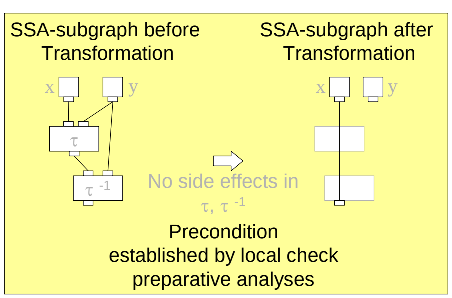
3

Algebraic Identity: Elimination of Operations and its Inverse



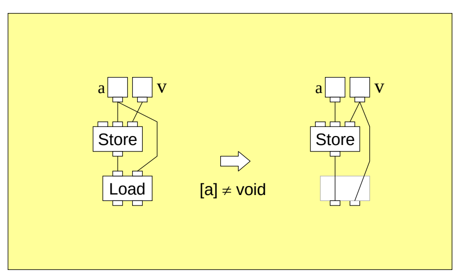
4

Graph Rewrite Schema



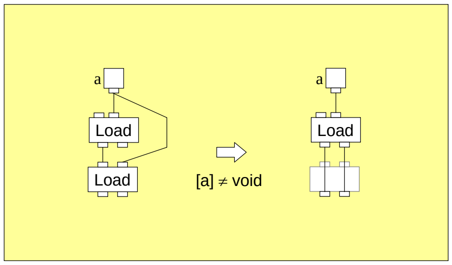
5

Elimination of Memory Operation and its Inverse

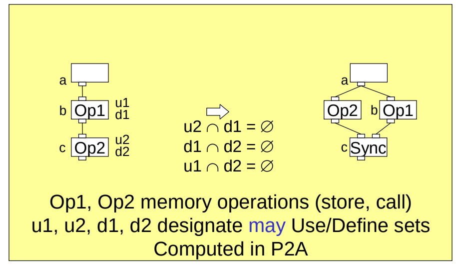


6

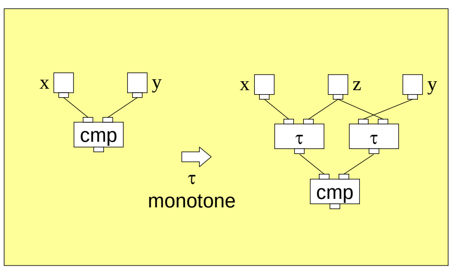
Elimination of Duplicated Memory Operations



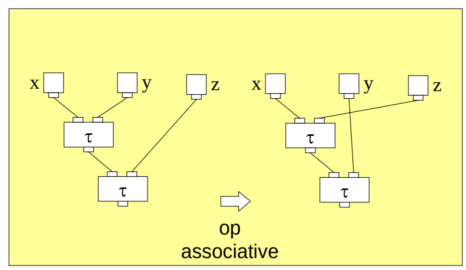
Elimination of non-essential dependencies



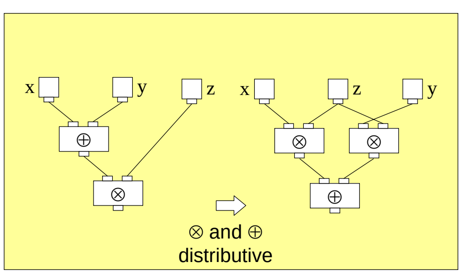
Algebraic Identity: Invariant Compares



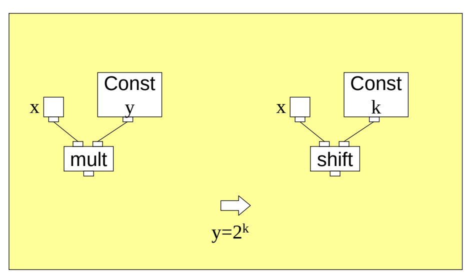
Associative Law



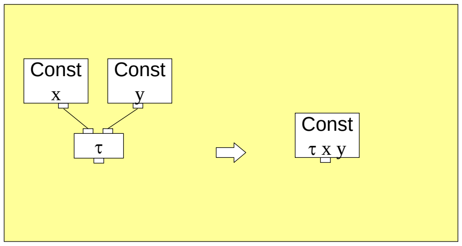
Distributive Law



Operator Simplification



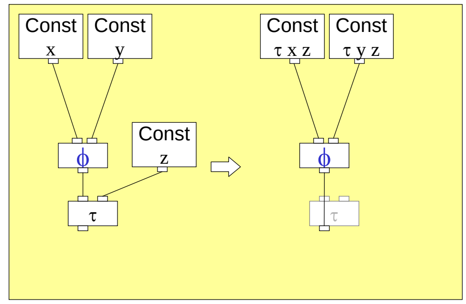
Constant Folding



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

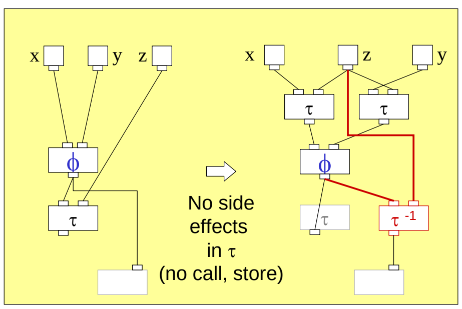
13

Constant folding over ϕ-functions



14

General: Moving arithmetic operations over ϕ-functions



15

Optimizations

- Strength reduction:
  - Bauer & Samelson 1959
  - Replace expensive by cheep 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

16

Example: Strength reduction

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

//Original loop body:
ao    = i*n*d + j*d
aij   = >a[0,0]<+ao
bo    = i*n*d + j*d
bij   = >b[0,0]<+bo
<bij> = <aij>
```

```
       adda=>a[0,0]<
       addb=>b[0,0]<
       d=4
       addend=adda+n*n*d;
LOOP:  jump(addend==adda) END
       <addb>=<adda>
       adda=adda+d
       addb=addb+d
       jump LOOP
END:   exit
```

17

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 iff 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++){…}`
  - Direct induction variable: `j`, implicit `j = j+1` (c= 1)
  - Indirect induction variable: `ao = i*n*d+j*d` (c'= `d`, c'' = `i*n*d`)
  - Note that `i*n*d` and `d` a loop constants for the inner loop

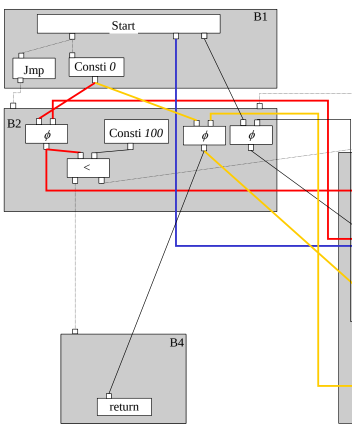
18

Induction Transformation Idea

- Transformation goal: values of induction variables should grow linearly with iteration; `add` operations replace `mult` operations
- Transformation:
  - Let i_0 initialization of i and induction variables, $i := i+c$ and $i' := c'*i+c''$
  - New variable ia initialized $ia := c' * i_0 + c''$
  - At loop end insert $ia = ia + c' * c$
  - Replace consistently operands i' by ia
  - Remove all assignments to i,i' and i,i' themselves if i is not used elsewhere (DFA)
- Example:
  - Before: `loop ao = i*n*d+j*d … j++ end loop`
  - After: `aoa = i*n*d loop … aoa = aoa + d end loop`

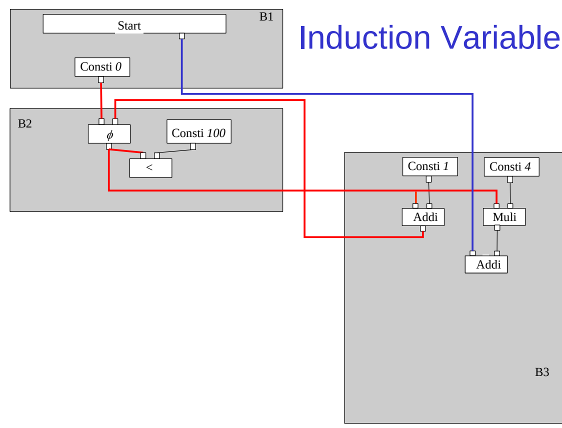
Induction Analysis: Implementation

- 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:
  - $i := i_x+c$ with loop constants c (direct induction variable)
  - $i := c*i_x+c_x$ with i_x induction variable and loop constants c, c' (indirect induction variable)
  - $i := \phi(i_1 \ldots i_n)$ with i_k 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

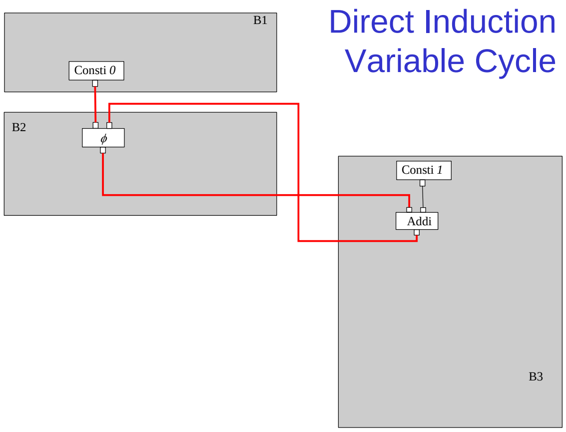



```
sum(array[int] a){
    s = 0;
    for(i=0;i<=100;i++){
        s:=a[i] + s;
    }
    return s;};
```

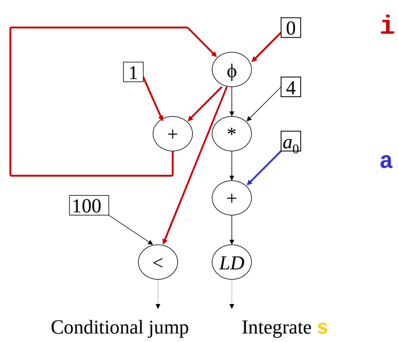
Induction Variables



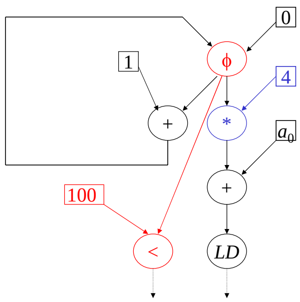
Direct Induction Variable Cycle



Induction Variables (Schematic)

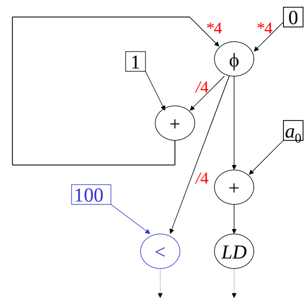


Move * over ϕ-function



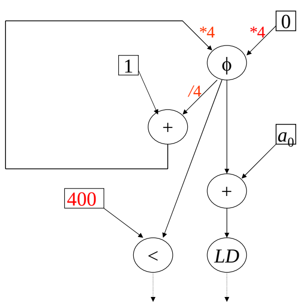
25

Move Multiplication



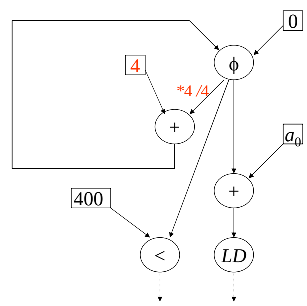
26

Invariant Compare



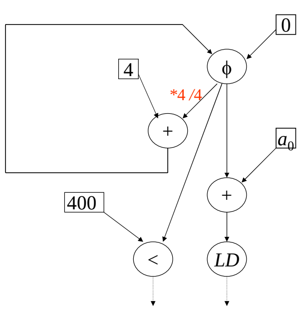
27

Distributive Law



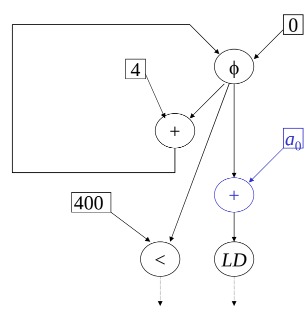
28

Operation and its Inverse



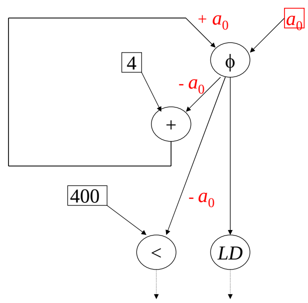
29

Move Addition



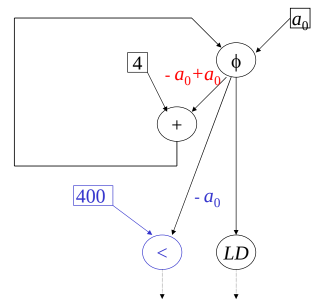
30

Move Addition



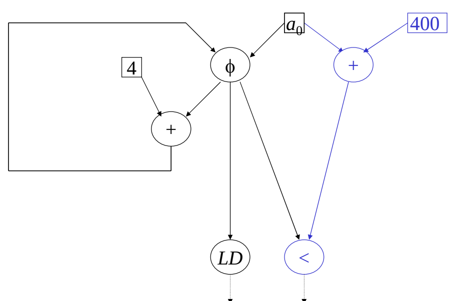
31

Associative Law

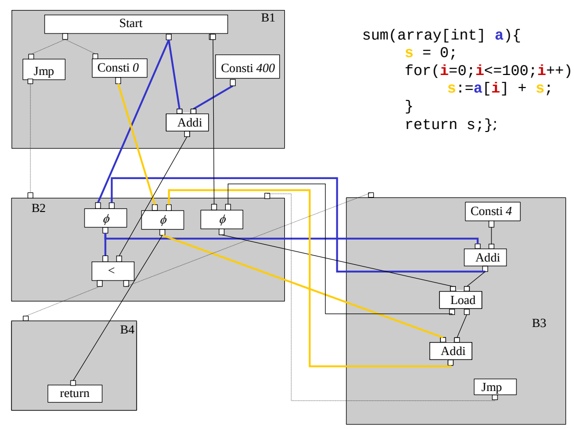


32

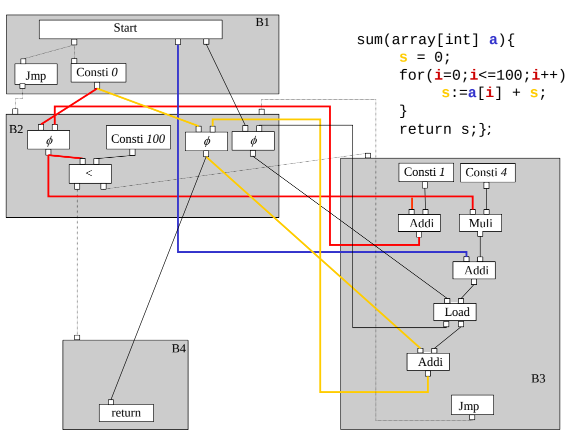
Change Compare



33




```
sum(array[int] a){
    s = 0;
    for(i=0;i<=100;i++){
        s:=a[i] + s;
    }
    return s;};
```




```
sum(array[int] a){
    s = 0;
    for(i=0;i<=100;i++){
        s:=a[i] + s;
    }
    return s;};
```

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

36

Observations on SSA

Operations that must be executed in original block:

1. ϕ-nodes,
2. Computations with exceptions
3. Jumps
4. “pinned” operations (postponed)

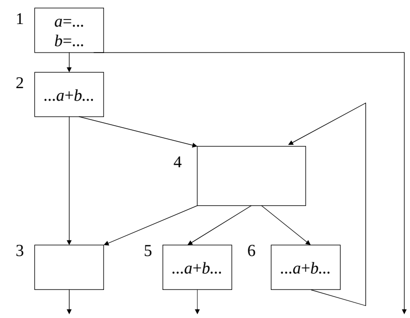
1. Observation:

   All other nodes could be computed in other blocks as well *iff* data dependencies are obeyed.

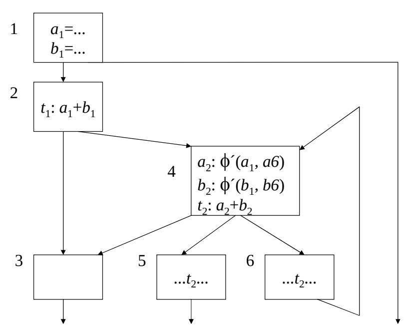
2. Observation:

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

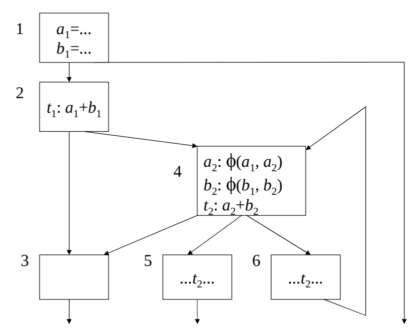
Example: Initial Situation



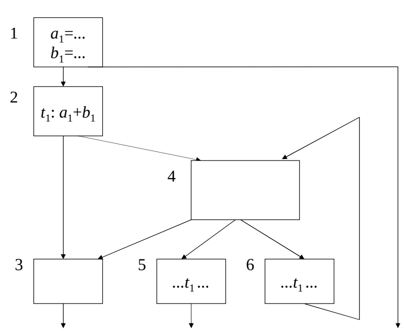
Immature ϕ'



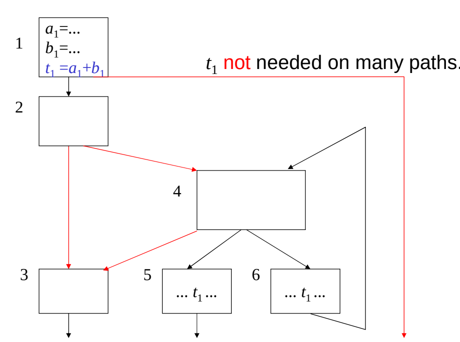
Mature $\phi' \rightarrow \phi$



Placement of computations

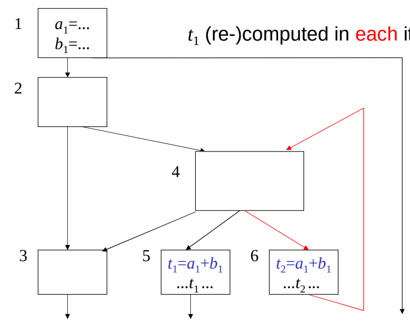


Placing t_1 *earliest*



Placing t_1 *latest*

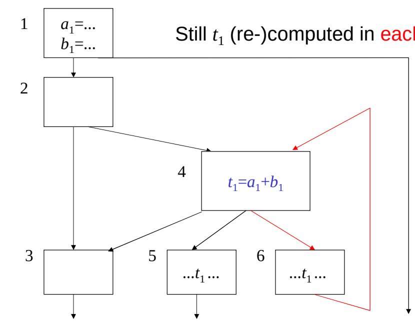
t_1 (re-)computed in each iteration.



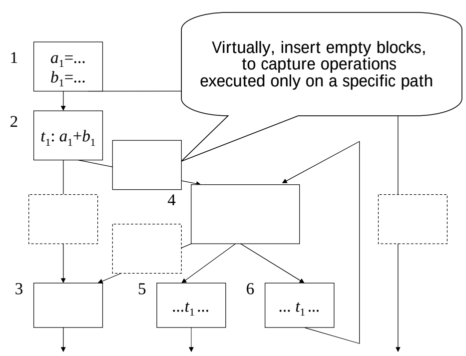
Placing t_1 *"optimally"*

According of Knoop, Steffen

Still t_1 (re-)computed in each iteration.

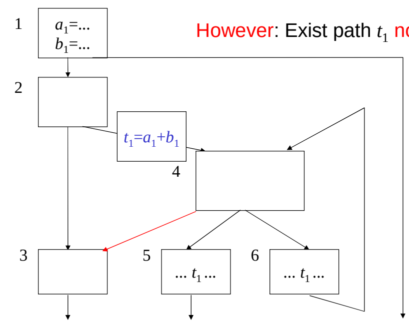


Insert Blocks



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
  - B contains operation τ computing t, B' contains τ' consuming t
  - If B' post-dominates B ($B' \leq B$) no dynamic redundancy
  - Assume all operations as partially (dynamically) redundant
  - For each operation τ computing t and the set of operations $\tau'_1 \ldots \tau'_n$ consuming t: if $B(\tau'_k) \leq B(\tau)$ for some k in $1 \ldots n$ then τ 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 τ
  - Move copies of τ towards the consuming operations $\tau'_1 \ldots \tau'_n$ 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 $t_2 : a_1 + b_1$ semantic equivalent computations at different positions (blocks)
  - Replace + by a „pinned" $\oplus_{Block}$
  - Thereby $\oplus$ operation additionally depends on the current block as new arguments
  - computations $a_1 \oplus_5 b_1$ and $a_1 \oplus_6 b_1$ not recognized as congruent any more

49

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 (array access, object layout)
- Parallelization
- …

50

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

51

Outline

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

52

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)

53

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)

54

Abstract values, addresses, memory

- References and values
  - allocation site lattice 2^{O} abstracts from objects O
  - arbitrary lattice X abstracts from values like `Integer` or `Boolean`
  - abstract heap memory M:
    - $O \times R \longrightarrow O$ (R set of fields with reference object semantics)
    - $O \times V \longrightarrow X$ (V set of fields with value semantics)
- Arrays
  - Treated as objects
  - abstract heap memory M:
    - $O \times R_{[]} \times I \longrightarrow O$ ($R_{[]}$ set of fields with type array of reference object)
    - $O \times V_{[]} \times I \longrightarrow X$ ($V_{[]}$ set of fields with type array of value)
    - I an arbitrary integer value latice (e.g., constant or power set lattice)
- Abstract address $Addr \subseteq 2^{O} \times F \times I$ (F set of field names)
  - object-field-(index) triples where index might be ignored

Updates of Memory

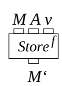
- Given an abstract object-field-(index) of a store operation
- In general, this abstract object points to more than one real memory cell
- A store operation overwrites only one of these cells, all others contain the same value
- Hence, a store to an abstract object-field-(index) adds a new possible (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:

$update(M, (o, n, \top), v) = v$ (if strong update possible)

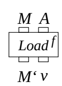
$= M(o, n, \top) \cup v$ (otherwise, weak update)

Transfer functions (insensitive, no arrays)

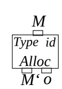
- $T_{store}(M, Addr, v) =$
  $M[a_1 \mapsto update(M, a_1, v)] \dots [a_k \mapsto update(M, a_k, v)]$
  $Addr = \{a_1 \dots a_k\}$ $a_i = (o_i, f, \top)$



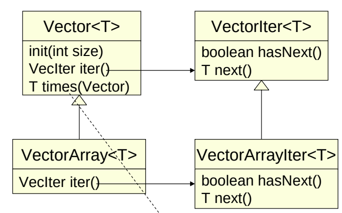
- $T_{load}(M, Addr) =$
  $(M, M(a_1) \cup \dots \cup M(a_k))$



- $T_{alloc(type)}(M) =$
  $(M[(o_{id}, n_1, \top) \mapsto \bot] \dots [(o_{id}, n_k, \top) \mapsto \bot], o_{id})$
  $\{n_1 \dots n_k\}$ attributes of $Type$

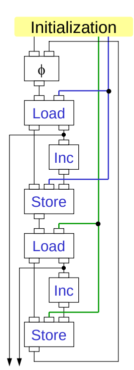


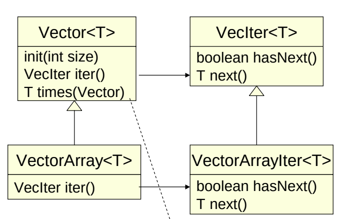
Example: Main Loop Inner Product Algorithm




```
T times(Vector v){
  i1 = iter();
  i2 = v.iter();
  for (s = 0; i1.hasNext(); )
    s = s+i1.next()*i2.next();
}
```


Example: SSA

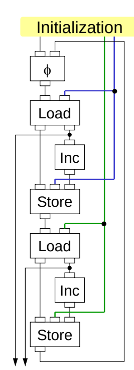


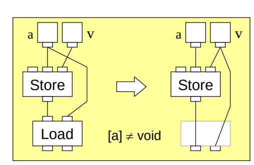



```
T times(Vector v){
  i1 = iter();
  i2 = v.iter();

  for (s = 0; i1.hasNext(); )
    s = s+ i1.next()*i2.next();
}
```

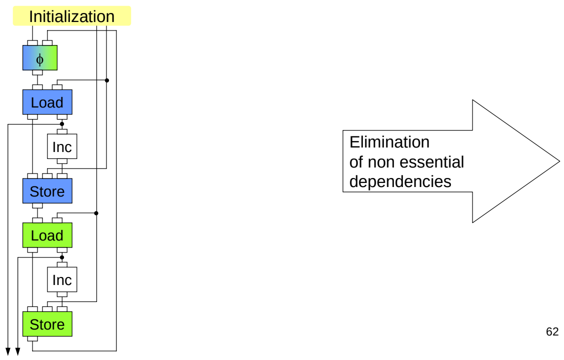

No provably different memory addresses





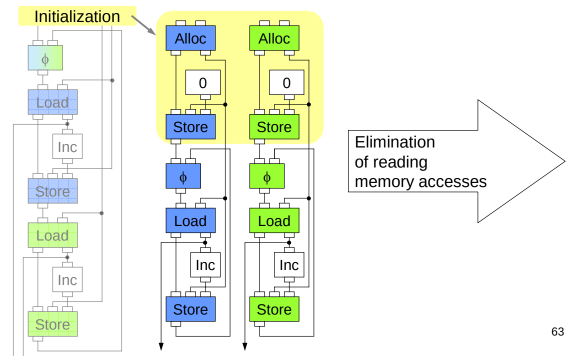
Not applicable!

Actually two Iterators?



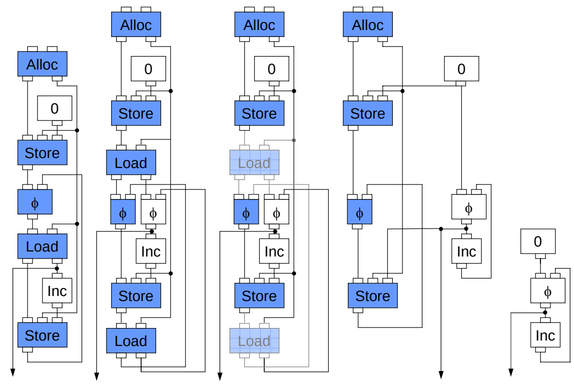
62

Initialization: disjoint memory guarantied

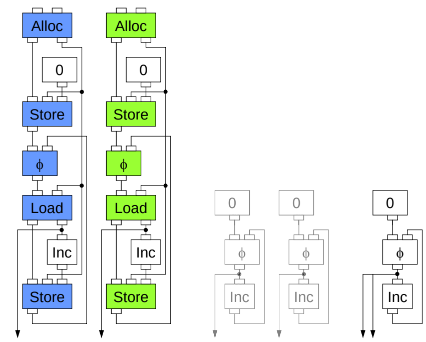


63

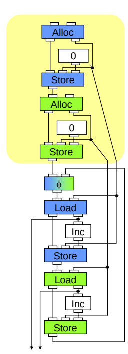
Memory objects replaced by values



Value numbering proofs equivalence

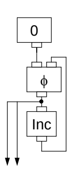


Example revisited



Optimization only possible due to joint application of single techniques:

- Global analysis
- Elimination of polymorphism
- Elimination of non essential dependencies
- Elimination von memory operations
- Traditional optimizations



66