Recitation Class
Compiler Design II

Laboratory for Software Technology
ETH Zurich

26.10.2007

Albert Noll

Outline

- Introduction to Optimization

- Presentation of Assignment 2
  – Constant Folding
  – Common Sub-Expression Evaluation
  – Copy Propagation
  – Warnings for Uninitialized Variables

Introduction to Optimization (1)

- optimization: rather *improve* performance than make code *optimal*
- it is formally un-decidable whether a particular optimization improves or worsens performance
- in general: optimizations should be as aggressive as possible BUT never produce incorrect results

Introduction to Optimization (2)

- 2 fundamental criteria to decide which optimization should be performed:
  - speed: increasing speed can either increase or decrease space
  - space: might be an important factor for limited systems with a small cache
- applied optimization also depends on:
  - language (object orientated)
  - architecture (register allocation for RISC architectures)

Which are the Most Important Optimizations?

- There are 4 groups
  - Group I: consists mostly of optimizations that operate on loops, but also others
    - example: constant folding, constant propagation, partial redundancy elimination, common sub-expression evaluation, loop invariant code motion,…
  - Group II: loop optimizations / are generally applicable to many programs
    - examples: local and global copy propagation, unnecessary bounds checking elimination, branch prediction, …

Which are the Most Important Optimizations?

- Group III: optimizations that apply to whole procedures or increase applicability of other optimizations:
  - examples: procedure integration, in-line expansion, scalar replacement of aggregates, …
- Group IV: optimizations that save code-space but generally do not save time
  - code hoisting, tail merging

Order of Optimizations

Scalar replacement of array references
Data-cache optimizations

A

for details see: *Advanced Compiler Design & Implementation*
Steven S. Muchnick

Order of Optimizations

Scalar replacement of array references
Data-cache optimizations

A

Procedure integration
Tail Call Optimization
Scalar Replacement of Aggregates
Sparse Conditional Constant Propagation
Inter-procedural constant Propagation
Procedure Specialization and Cloning
Sparse Conditional Constant Propagation

B

Order of Optimizations

Scalar replacement of array references
Data-cache optimizations

A

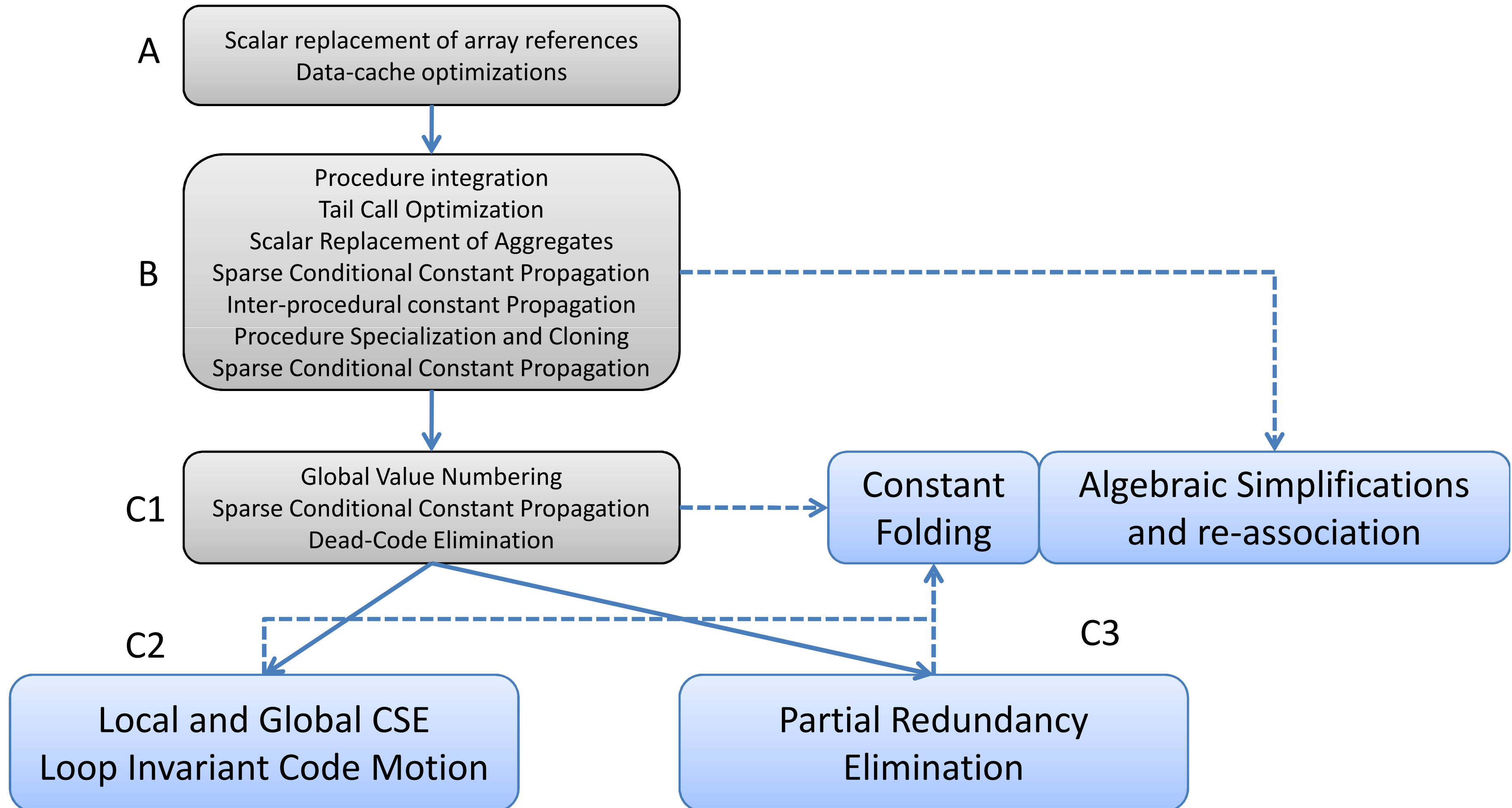
Procedure integration
Tail Call Optimization
Scalar Replacement of Aggregates
Sparse Conditional Constant Propagation
Inter-procedural constant Propagation
Procedure Specialization and Cloning
Sparse Conditional Constant Propagation

B

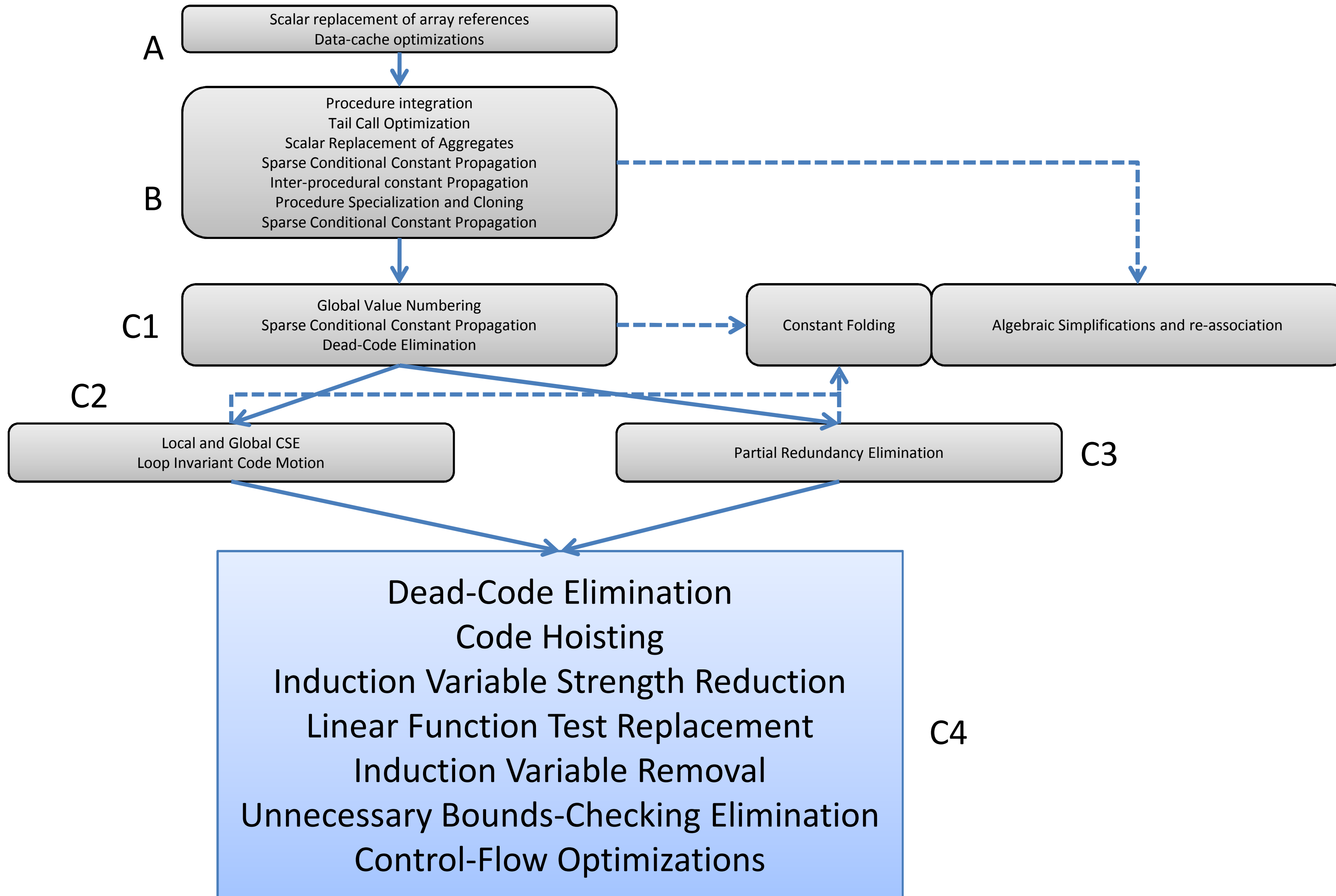
Global Value Numbering
Sparse Conditional Constant Propagation
Dead-Code Elimination

C1

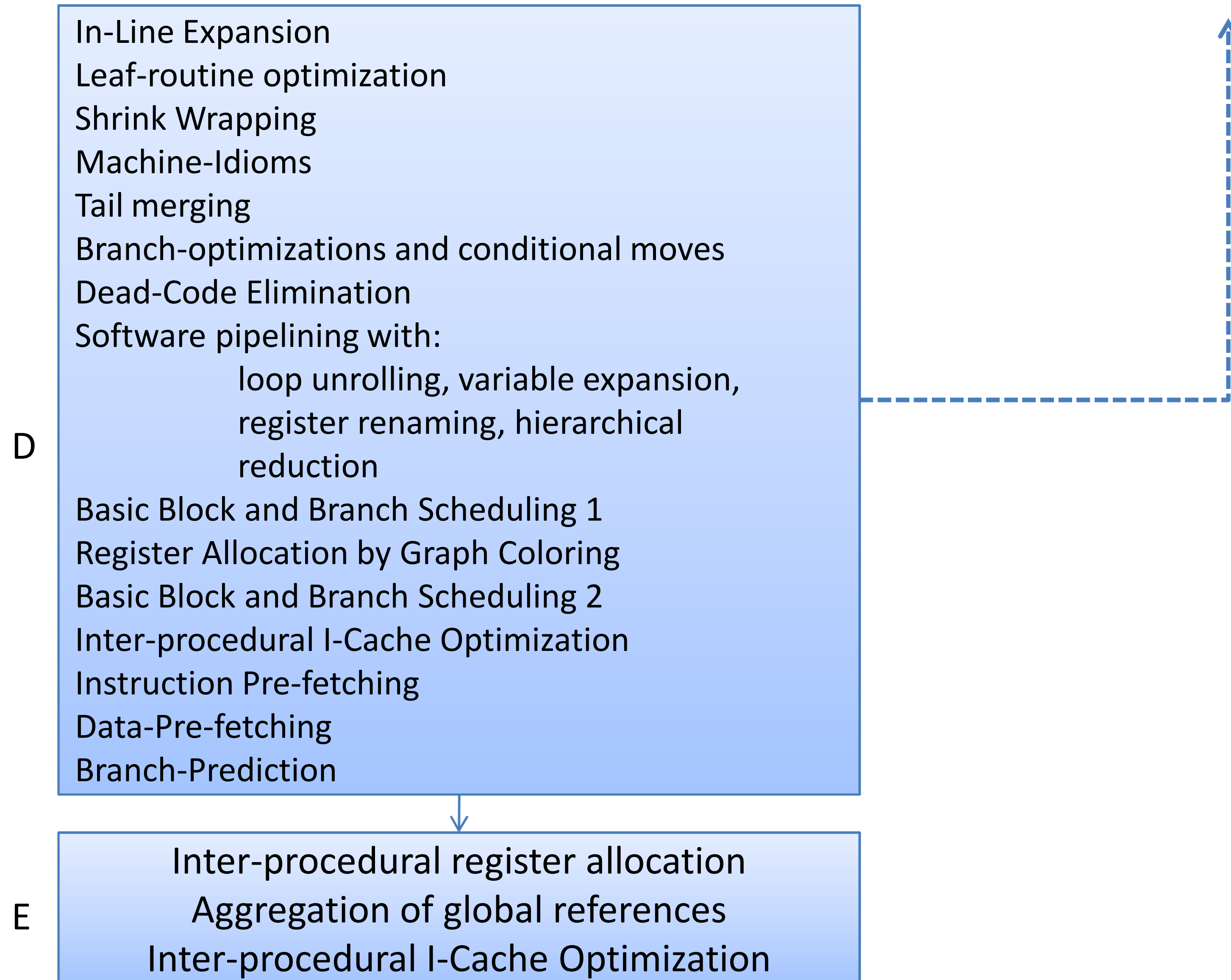
Order of Optimizations

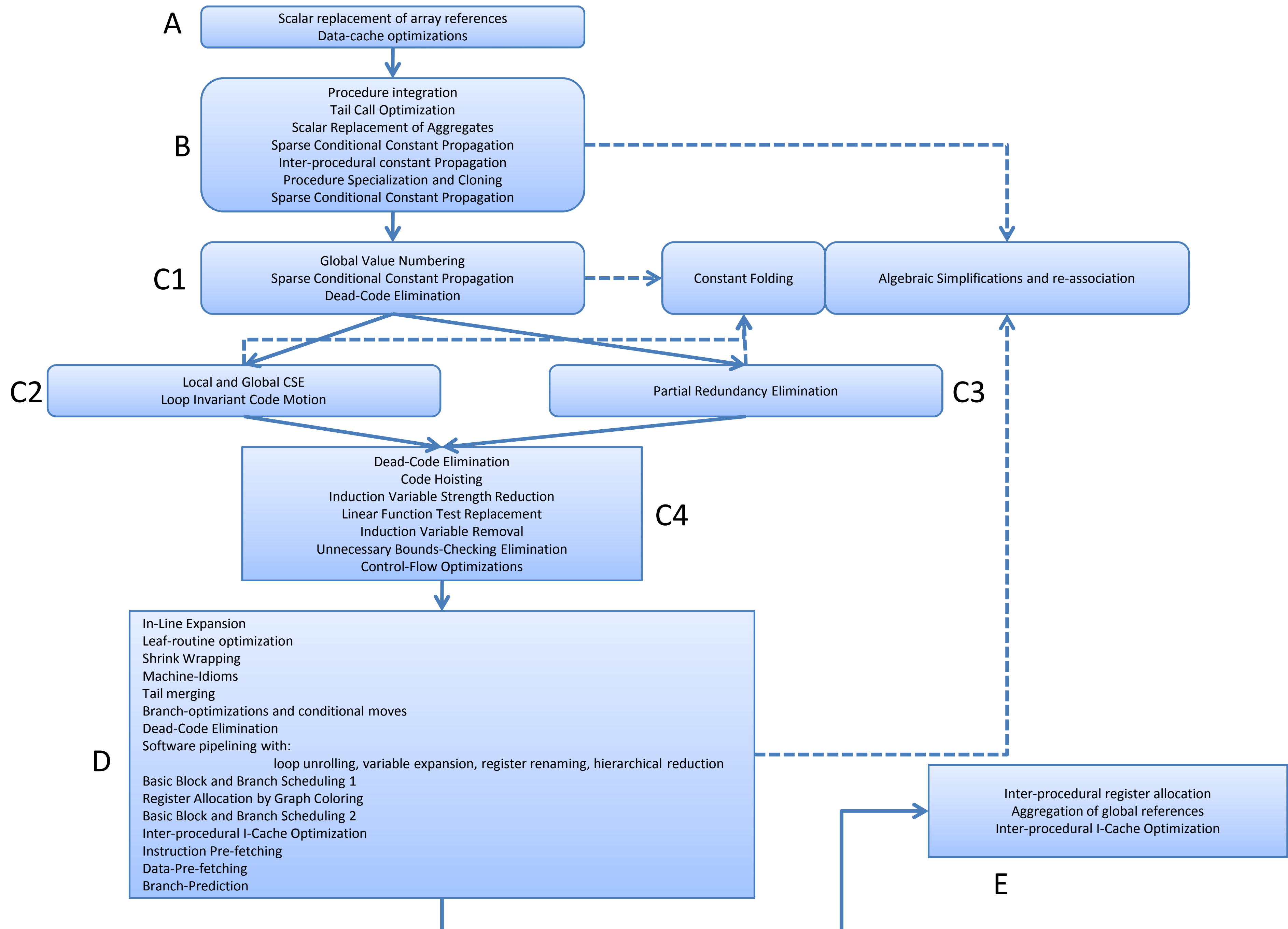


Order of Optimizations



Order of Optimizations




A
Scalar replacement of array references
Data-cache optimizations
B
Procedure integration
Tail Call Optimization
Scalar Replacement of Aggregates
Sparse Conditional Constant Propagation
Inter-procedural constant Propagation
Procedure Specialization and Cloning
Sparse Conditional Constant Propagation
C1
Global Value Numbering
Sparse Conditional Constant Propagation
Dead-Code Elimination
Constant Folding
Algebraic Simplifications and re-association
C2
Local and Global CSE
Loop Invariant Code Motion
Partial Redundancy Elimination
C3
Dead-Code Elimination
Code Hoisting
Induction Variable Strength Reduction
Linear Function Test Replacement
Induction Variable Removal
Unnecessary Bounds-Checking Elimination
Control-Flow Optimizations
C4
In-Line Expansion
Leaf-routine optimization
Shrink Wrapping
Machine-Idioms
Tail merging
Branch-optimizations and conditional moves
Dead-Code Elimination
Software pipelining with:
loop unrolling, variable expansion, register renaming, hierarchical reduction
Basic Block and Branch Scheduling 1
Register Allocation by Graph Coloring
Basic Block and Branch Scheduling 2
Inter-procedural I-Cache Optimization
Instruction Pre-fetching
Data-Pre-fetching
Branch-Prediction
D
Inter-procedural register allocation
Aggregation of global references
Inter-procedural I-Cache Optimization
E

Constant Folding

- General idea:
  - evaluate constant expressions at compile-time and replace the constant expressions by their values
  - Example

```
int foo() {
    int i = 1, j = 2;
    int k = i+j;                    // replace k by three
    return k;
}
```

Constant Folding

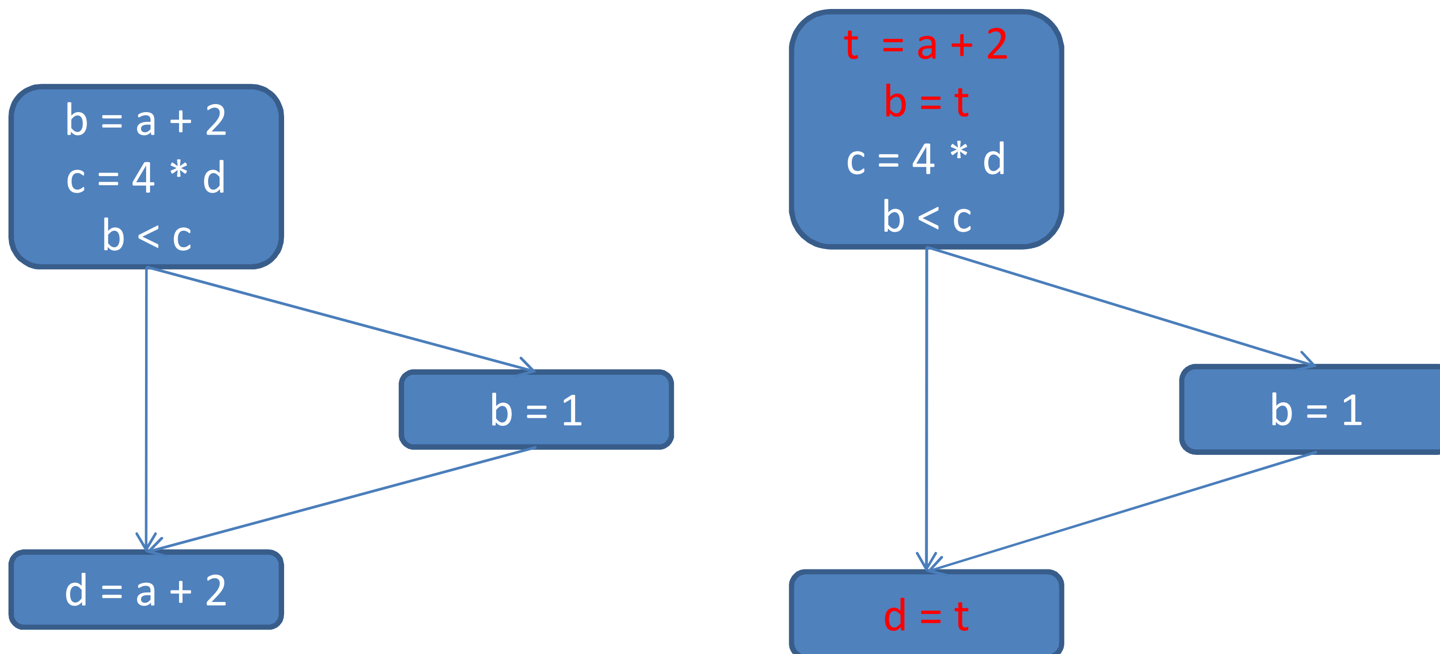
- Possible Problems:
  - division by zero
  - overflows, underflows
  - expressions should be evaluated the same way at compile time as they are at runtime

Common Sub-Expression Evaluation

- "An expression in a program is a ***common sub-expression*** if there is another occurrence of the expression whose evaluation precedes this one in execution order and **if the operands of the expression remain unchanged between two evaluations**"
- In SSA we can determine easily if the operands were changed

Common Sub-Expression Evaluation

- **Common Sub-Expression Evaluation** removes re-computations of common sub-expressions and replaces them with uses of results



Common Sub-Expression Evaluation

- Is this always worthwhile?
- No – but why?
- for example: due to a little number of available registers the temporary result has to be stored in memory => load instruction might be more expensive than original instruction

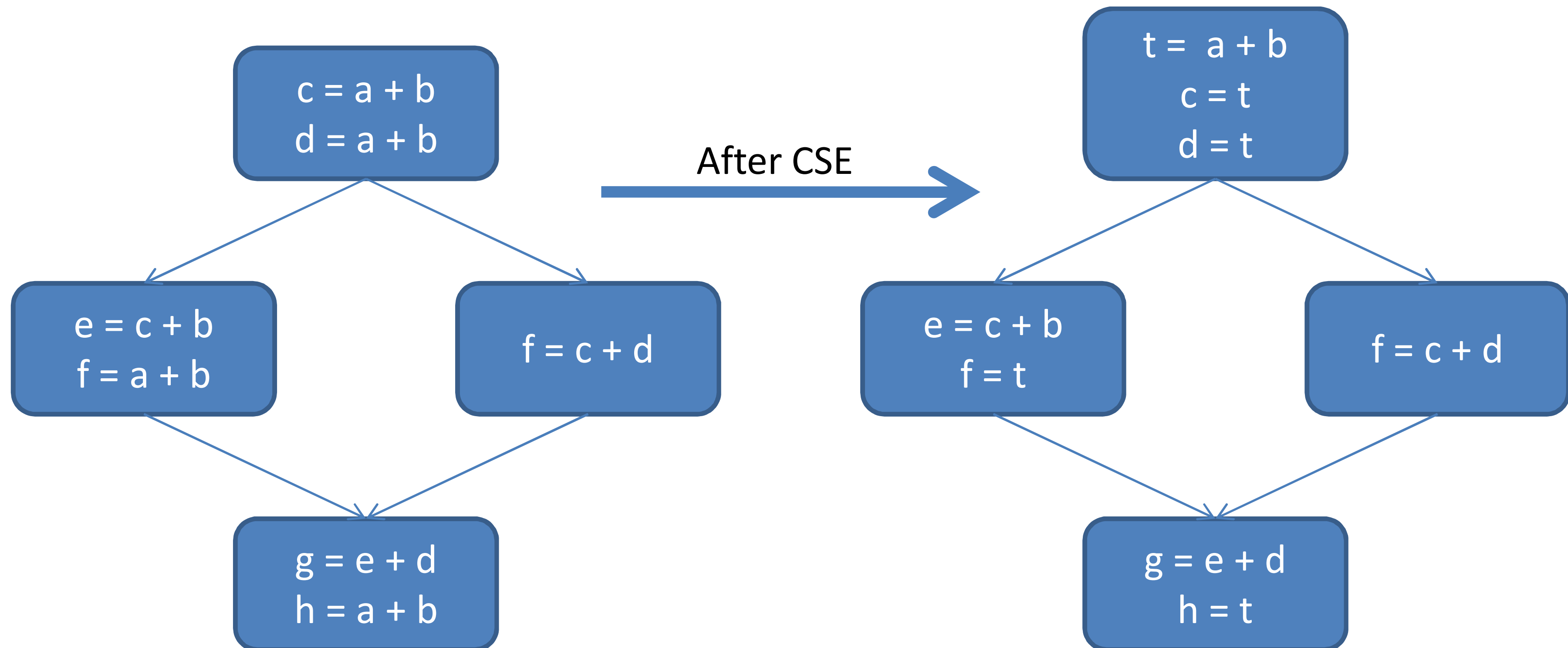
Common Sub-Expression Evaluation

- For the Assignment:
  - mandatory: replace only those expressions, that are ‘obviously’ equal:

```
int c = a + b;
int d = a + b;          //mandatory
int e = b + a;          //not mandatory
```

Copy Propagation

- An example:



Copy Propagation