Compiler Construction

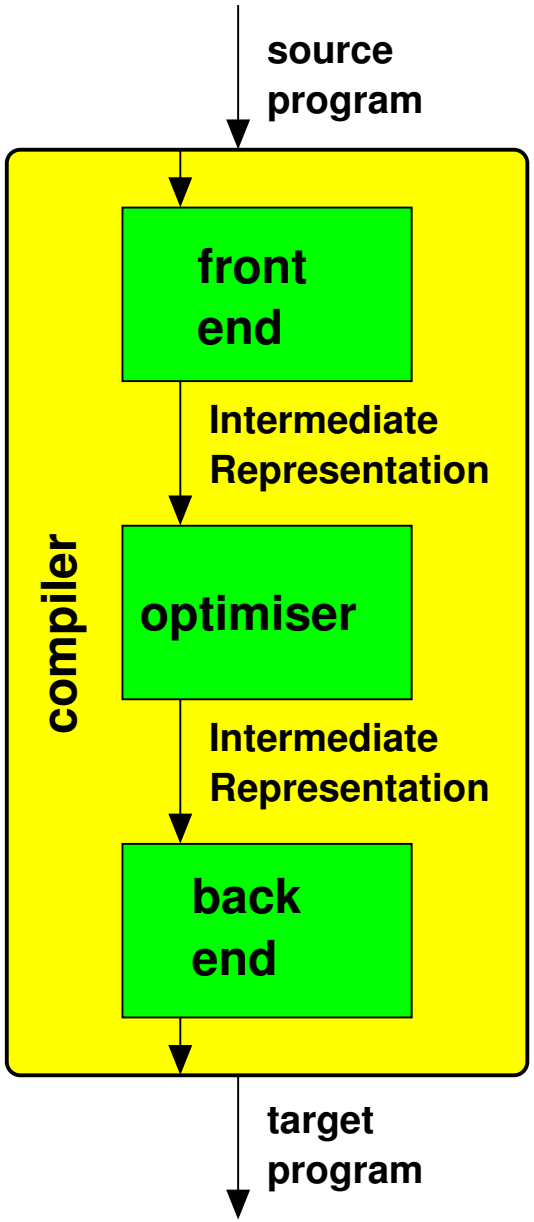
Chapter 6

Machine-Independent Optimisation

Clemens Grelck

System and Network Engineering Lab

Advanced Compiler Structure



Front end:

- Analysis of source program

Optimiser (Middle end):

- Code transformation within intermediate representation
- Motivated by some non-functional requirements / desires:
- Speed, power, size, ...

Back end:

- Synthesis of target program

Machine-Independent Optimisation ?

Technique:

- Replacement of Code within intermediate representation
  - context-free
  - context-sensitive

Machine-Independent Optimisation ?

Technique:

- Replacement of Code within intermediate representation
  - context-free
  - context-sensitive

Motivation:

- New code is “better” than original code
  - (presumed) execution speed
  - code size
  - (presumed) energy consumption

Machine-Independent Optimisation ?

Technique:

- Replacement of Code within intermediate representation
  - context-free
  - context-sensitive

Motivation:

- New code is "better" than original code
  - (presumed) execution speed
  - code size
  - (presumed) energy consumption

Requirement:

- New code is semantically equivalent to original code
  - under all possible circumstances
  - and not just "mostly"

Machine-Independent Optimisation ?

Machine-independent optimisation:

- Source code oriented
- Reduce abstractions in general
- No particular assumption on execution hardware
- Very general model of execution machinery

Machine-Independent Optimisation ?

Machine-independent optimisation:

- Source code oriented
- Reduce abstractions in general
- No particular assumption on execution hardware
- Very general model of execution machinery

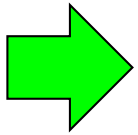
Machine-dependent optimisation:

- Target code oriented
- Adapt generated code to execution environment
- Exploit specific hardware features

Optimisation: Algebraic Simplification

Example:

```
int x,y;
...
y = ... x*1 ... ;
```



```
int x,y;
...
y = ... x ... ;
```

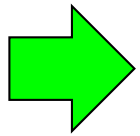
Characteristics:

- We exploit algebraic properties of operations
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- Good idea on any machine
- Simple: context-free transformation
- Trade-off: none

Optimisation: Algebraic Simplification

Example:

```
int x,y;
...
y = ... x*0 ... ;
```



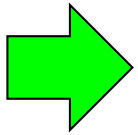
```
int x,y;
...
y = ... 0 ... ;
```

Correct ?

Optimisation: Algebraic Simplification

Example:

```
int x,y;
...
y = ... x*0 ... ;
```



```
int x,y;
...
y = ... 0 ... ;
```

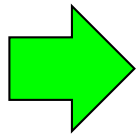
Same characteristics:

- We exploit algebraic properties of operations
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- Good idea on any machine
- Simple: context-free transformation
- Trade-off: none

Optimisation: Algebraic Simplification

Another Example:

```
int x,y;
...
y = (...) * 0 ;
```



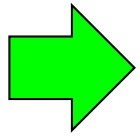
```
int x,y;
...
y = 0;
```

Correct ?

Optimisation: Algebraic Simplification

Another Example:

```
int x,y;
...
y = (...) * 0 ;
```



```
int x,y;
...
y = 0;
```

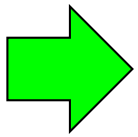
Same Characteristics (?):

- We likewise exploit algebraic properties of operations
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- Good idea on any machine
- Simple: context-free transformation
- Trade-off: none
- BUT: Is the transformation meaning-preserving? Under all circumstances?

Optimisation: Strength Reduction

Example:

```
int x,y;
...
y = ... x * 2 ... ;
```



```
int x,y;
...
y = ... x + x... ;
```

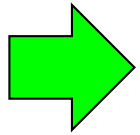
Characteristics:

- Based on the assumption that addition is cheaper than multiplication
- Same number of operations in compiled code
- Cheap operation instead of expensive one to execute ($\rightarrow$ speed, energy)
- Only semi machine independent

Optimisation: Constant Folding

Example:

```
int x,y;
...
y = ... 2 + 4 ... ;
```



```
int x,y;
...
y = ... 6 ... ;
```

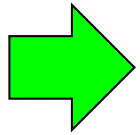
Characteristics:

- Compute operation at compile time if operands are known
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- Good idea on any machine
- Simple: context-free transformation
- Trade-off: none

Optimisation: Constant Folding

Example:

```
int x,y;
...
y = ... 2 + 4 ... ;
```



```
int x,y;
...
y = ... 6 ... ;
```

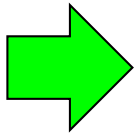
Characteristics:

- Compute operation at compile time if operands are known
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- Good idea on any machine
- Simple: context-free transformation
- Trade-off: none
- Careful: Be sure that compile time operation is **identical** to runtime operation !

Optimisation: Variable Propagation

Example:

```
int x,y,z;
...
x = ... ;
y = x ;
z = y ... y ...y ;
```



```
int x,y,z;
...
x = ... ;
z = x ... x ... x ;
```

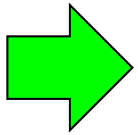
Characteristics:

- Eliminate variable to variable assignment (chains)
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- One variable less ($\rightarrow$ register pressure)
- Good idea on any machine
- Trade-off: none

Optimisation: Constant Propagation

Example:

```
int x,y;
...
x = 42 ;
y = x ... x ...x ;
```



```
int y;
...
y = 42 ... 42 ... 42 ;
```

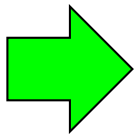
Characteristics:

- Eliminate variable to variable assignment chains
- One operation less in compiled code ($\rightarrow$ code size)
- One operation less to execute ($\rightarrow$ speed, energy)
- One variable less ($\rightarrow$ register pressure)
- Good idea on any machine provided constant register initialisation is not more expensive than register transfer
- Good idea on (almost) any machine

Optimisation: Dead Code Elimination

Example:

```
int x;
...
x = fib(12) ;
return 42 ;
```



```
...
return 42 ;
```

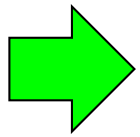
Characteristics:

- Eliminate code that does not contribute to result
- Less instructions in compiled code ($\rightarrow$ code size)
- Less instructions to execute ($\rightarrow$ speed, energy)
- Less variables ($\rightarrow$ register pressure)
- Good idea on any machine
- Trade-off: none

Optimisation: Dead Code Elimination

Example:

```
int x;
...
x = fib(12) ;
return 42 ;
```



```
...
return 42 ;
```

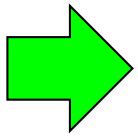
Characteristics:

- Eliminate code that does not contribute to result
- Less instructions in compiled code ($\rightarrow$ code size)
- Less instructions to execute ($\rightarrow$ speed, energy)
- Less variables ($\rightarrow$ register pressure)
- Good idea on any machine
- Trade-off: none
- **BUT: Do we know fib is free of side effects ?**

Optimisation: Common Subexpression Elimination

Example:

```
int y,z,m,n,o;
...
m = 2 * y * z;
n = 3 * y * z;
o = 2 * y - z;
```



```
int y,z,m,n,o, t;
...
t = 2 * y;
m = t * z;
n = 3 * y * z;
o = t - z;
```

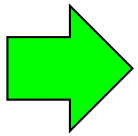
Characteristics:

- Identify identical (sub-)expressions
- Precompute (sub-)expression once
- Assign result to fresh variable
- Less operations in compiled code ($\rightarrow$ code size)
- Less operations to execute ($\rightarrow$ speed, energy)

Optimisation: Common Subexpression Elimination

Example:

```
int y,z,m,n,o;
...
m = 2 * y * z;
n = 3 * y * z;
o = 2 * y - z;
```



```
int y,z,m,n,o, t;
...
t = 2 * y;
m = t * z;
n = 3 * y * z;
o = t - z;
```

Characteristics:

- Identify identical (sub-)expressions
- Precompute (sub-)expression once
- Assign result to fresh variable
- Less operations in compiled code ($\rightarrow$ code size)
- Less operations to execute ($\rightarrow$ speed, energy)
- **BUT: space/time trade-off:**
  memory needed to store intermediate value

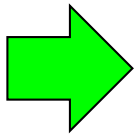
Code Order Matters

Observation:

- Effectiveness of CSE, etc, very much depends on **exact** intermediate representation.
- Small variations in code can have huge effect.

Example:

```
int y,z,m,n,o;
...
m = 2 * (y * z);
n = 3 * (y * z);
o = 2 * y - z;
```



```
int y,z,m,n,o, t;
...
t = y * z;
m = 2 * t;
n = 3 * t;
o = 2 * y - z;
```

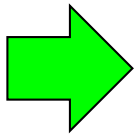
Code Order Matters

Observation:

- Effectiveness of CSE, etc, very much depends on **exact** intermediate representation.
- Small variations in code can have huge effect.

Example:

```
int y,z,m,n,o;
...
m = 2 * (y * z);
n = 3 * (y * z);
o = 2 * y - z;
```



```
int y,z,m,n,o, t;
...
t = y * z;
m = 2 * t;
n = 3 * t;
o = 2 * y - z;
```

Example:

- Small change in associativity yields entirely different code.

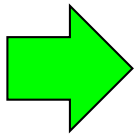
Code Order Matters

Observation:

- Effectiveness of CSE, etc very much depend on exact intermediate representation.
- Small variations in code can have huge effect.

Example:

```
int y,z,m,n,o;
...
m = 2 * (y * z);
n = (3 * y) * z;
o = 2 * y - z;
```



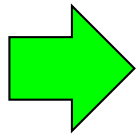
Code Order Matters

Observation:

- Effectiveness of CSE, etc very much depend on exact intermediate representation.
- Small variations in code can have huge effect.

Example:

```
int y,z,m,n,o;
...
m = 2 * (y * z);
n = (3 * y) * z;
o = 2 * y - z;
```



Observation:

- Semantically identical codes may have very different intermediate representations.

Code Order Matters

Observation:

- Effectiveness of CSE, etc very much depends on **exact** intermediate representation.
- Small variations in code can have huge effect.

Insight:

- An effective Common Subexpression Elimination must take operator associativity and commutativity into account !

Code Order Matters

Observation:

- Effectiveness of CSE, etc very much depends on **exact** intermediate representation.
- Small variations in code can have huge effect.

Insight:

- An effective Common Subexpression Elimination must take operator associativity and commutativity into account !

Question:

- Can these capabilities be expanded to cover user-defined functions ?

A Taxonomy of Optimisations

Local Optimisation:

- Scope of optimisation restricted to single **basic block**
- Code transformation focuses on expression evaluation
- Examples: all optimisations so far

What is a basic block ?

- Sequence of statements (in an imperative language)
- All possible control flows enter at top
- All possible control flows leave at bottom
- No control flow divergence within sequence

A Taxonomy of Optimisations

Local Optimisation:

- Scope of optimisation restricted to single basic block
- Code transformation focuses on expression evaluation
- Examples: all optimisations so far

Intra-procedural optimisation:

- Scope of optimisation restricted to individual procedure
- Prevailing motivation: improve control flow
- Examples: loop optimisations

A Taxonomy of Optimisations

Local Optimisation:

- Scope of optimisation restricted to single basic block
- Code transformation focuses on expression evaluation
- Examples: all optimisations so far

Intra-procedural optimisation:

- Scope of optimisation restricted to individual procedure
- Prevailing motivation: improve control flow
- Examples: loop optimisations

Inter-procedural optimisation:

- Whole program view
- In the focus: procedure calls
- Examples: function inlining, specialisation

Control Flow Graph

Definition:

- Intermediate representation that makes control flow explicit
- Nodes of CFG are basic blocks
- Arcs of the CFG explicate control flow between basic blocks

Control Flow Graph

Definition:

- Intermediate representation that makes control flow explicit
- Nodes of CFG are basic blocks
- Arcs of the CFG explicate control flow between basic blocks

Example:

```
c = 2 * a + 5;
d = c - e;
if (d<0) {
 d = 0;
}
else {
 d = d / 2;
}
foo(i);
```

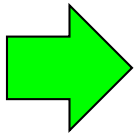
Control Flow Graph

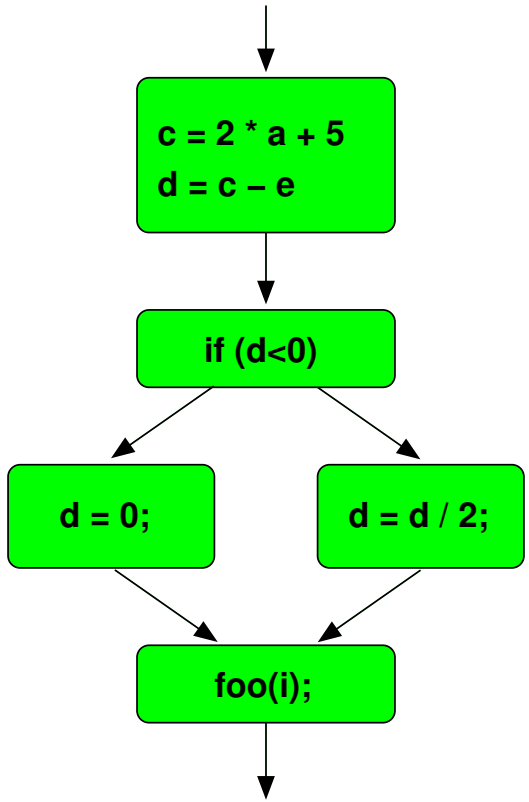
Definition:

- Intermediate representation that makes control flow explicit
- Nodes of CFG are basic blocks
- Arcs of the CFG explicate control flow between basic blocks

Example:

```
c = 2 * a + 5;
d = c - e;
if (d<0) {
 d = 0;
}
else {
 d = d / 2;
}
foo(i);
```





Control Flow Graph

Definition:

- Intermediate representation that makes control flow explicit
- Nodes of CFG are basic blocks
- Basic block: sequence of statements with linear control flow
- Arcs of the CFG explicate control flow between basic blocks

Example:

```
c = 2 * a + 5;
d = c - e;
i = 0;
while (d>1) {
 d = d / 2;
 i = i + 1;
}
foo(i);
```

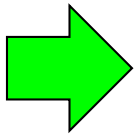
Control Flow Graph

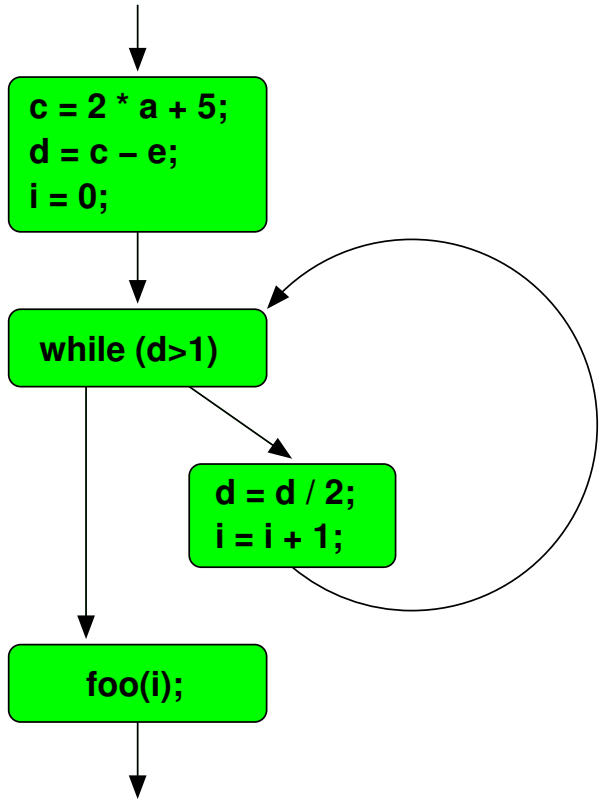
Definition:

- Intermediate representation that makes control flow explicit
- Nodes of CFG are basic blocks
- Basic block: sequence of statements with linear control flow
- Arcs of the CFG explicate control flow between basic blocks

Example:

```
c = 2 * a + 5;
d = c - e;
i = 0;
while (d>1) {
 d = d / 2;
 i = i + 1;
}
foo(i);
```





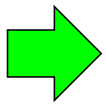
Optimisation: Loop Interchange

Characteristic:

- Exchange pairs of loops in a loop nest
- Caution: This changes the order of iterations !

Example:

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



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

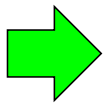
Optimisation: Loop Interchange

Characteristic:

- Exchange pairs of loops in a loop nest
- Caution: This changes the order of iterations !

Example:

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



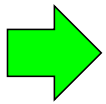
```
for (int i = 0, m) {
 for (int j = 0, n) {
  a[i,j] = b[i,j] + c[i,j];
 }
}
```

Motivation ?

Optimisation: Loop Interchange

Example:

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



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

Motivation:

- Modern memory hierarchies favour unit-stride access to arrays.
- Every memory load operation retrieves a whole cache line of adjacent addresses
- Successive array references alongside the array's storage pattern (here row-major) is much more efficient than doing the same against the array's storage pattern.

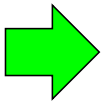
Optimisation: Loop Unrolling

Characteristic:

- Identify exact trip count n of loop
- Replace loop by n successive copies of loop body
- Replace occurrences of induction variable by constants

Example:

```
n = 4;
for (int i = 0, m) {
 for (int j = 0, n) {
  a[i,j] = b[i,j] + c[i,j];
 }
}
```



```
for (int i = 0, m) {
 a[i,0] = b[i,0] + c[i,0];
 a[i,1] = b[i,1] + c[i,1];
 a[i,2] = b[i,2] + c[i,2];
 a[i,3] = b[i,3] + c[i,3];
}
```

Optimisation: Loop Unrolling

Motivation:

- Avoid loop overhead
- Create larger basic block
- Loop's induction variable becomes constant
- $\rightarrow$ triggers further optimisation
- Trade-off: Larger code size

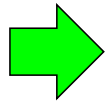
Optimisation: Loop Unrolling

Motivation:

- Avoid loop overhead
- Create larger basic block
- Loop's induction variable becomes constant
- $\rightarrow$ triggers further optimisation
- Trade-off: Larger code size

Example: offset computations:

```
for (int i = 0, m) {
 a[i,0] = b[i,0] + c[i,0];
 a[i,1] = b[i,1] + c[i,1];
 a[i,2] = b[i,2] + c[i,2];
 a[i,3] = b[i,3] + c[i,3];
}
```

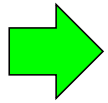


```
for (int i = 0, m) {
 a[i*4+0] = b[i*4+0] + c[i*4+0];
 a[i*4+1] = b[i*4+1] + c[i*4+1];
 a[i*4+2] = b[i*4+2] + c[i*4+2];
 a[i*4+3] = b[i*4+3] + c[i*4+3];
}
```

Optimisation: Loop Unrolling

Example: common subexpression elimination:

```
for (int i = 0, m) {
 a[i*4+0] = b[i*4+0] + c[i*4+0];
 a[i*4+1] = b[i*4+1] + c[i*4+1];
 a[i*4+2] = b[i*4+2] + c[i*4+2];
 a[i*4+3] = b[i*4+3] + c[i*4+3];
}
```



```
for (int i = 0, m) {
 _t0 = i*4;
 _t1 = _t0+1;
 _t2 = _t0+2;
 _t3 = _t0+3;
 a[_t0] = b[_t0] + c[_t0];
 a[_t1] = b[_t1] + c[_t1];
 a[_t2] = b[_t2] + c[_t2];
 a[_t3] = b[_t3] + c[_t3];
}
```

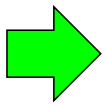
Optimisation: Loop Unswitching

Characteristics:

- Conditional within counted loop
- Predicate refers to loop's induction variable
- Loop unswitching avoids predicate evaluation and jump in every iteration of outer loop

Example:

```
for (int i = 0, 100) {
  if (i == p) {
    do_this(...);
  }
  else {
    do_that(...);
  }
}
```

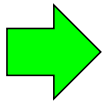


```
for (int i = 0, p) {
  do_that(...);
}

do_this(...);

for (int i = p+1, 100) {
  do_that(...);
}
```

Optimisation: Loop Unswitching

Example:

```
for (int i = 0, 100) {
  if (i == p) {
    do_this(...);
  }
  else {
    do_that(...);
  }
}
```



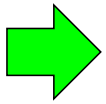
```
for (int i = 0, p) {
  do_that(...);
}

do_this(...);

for (int i = p+1, 100) {
  do_that(...);
}
```

Do you spot the error ?

Optimisation: Loop Unswitching

Example:

```
for (int i = 0, 100) {
  if (i == p) {
    do_this(...);
  }
  else {
    do_that(...);
  }
}
```



```
for (int i = 0, p) {
  do_that(...);
}

do_this(...);

for (int i = p+1, 100) {
  do_that(...);
}
```

Do you spot the error ?

What if p is not in between 0 and 100 ?

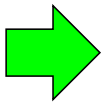
Optimisation: Loop Unswitching

Characteristics:

- Conditional within counted loop
- Predicate refers to loop's induction variable
- Loop unswitching avoids predicate evaluation and jump in every iteration of outer loop

Example:

```
for (int i = 0, 100) {
  if (i == p) {
    do_this(...);
  }
  else {
    do_that(...);
  }
}
```



```
for (int i = 0, min(p,100)) {
  do_that(...);
}

if (0<=p && p<100) {
  do_this(...);
}

for (int i = max(p+1,0), 100) {
  do_that(...);
}
```

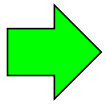
Optimisation: Strip Mining

Characteristics:

- Single loop is replaced by nesting of two loops
- Outer loop has a fixed compiler-determined step
- Inner loop ranges between zero and that step
- All occurrences of original induction variable are replaced by sum of the two new induction variables

Example:

```
for (int i = 1, m-1) {
 b[i] = a[i-1]
      + a[i]
      + a[i+1];
}
```



```
for (int i = 1, m-1, 4) {
 for (int j = 0, 4) {
  b[i+j] = a[(i+j)-1]
         + a[i+j]
         + a[(i+j)+1];
 }
}
```

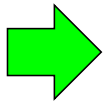
Optimisation: Strip Mining

Characteristics:

- Single loop is replaced by nesting of two loops
- Outer loop has a fixed compiler-determined step
- Inner loop ranges between zero and that step
- All occurrences of original induction variable are replaced by sum of the two new induction variables

Example:

```
for (int i = 1, m-1) {
 b[i] = a[i-1]
      + a[i]
      + a[i+1];
}
```



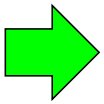
```
for (int i = 1, m-1, 4) {
 for (int j = 0, 4) {
  b[i+j] = a[(i+j)-1]
         + a[i+j]
         + a[(i+j)+1];
 }
}
```

Meaning-preserving ??

Optimisation: Strip Mining

Example:

```
for (int i = 1, m-1) {
 b[i] = a[i-1]
      + a[i]
      + a[i+1];
}
```



```
strip = 4;
lower = 1;
upper = m-1;
range = upper - lower;
split = upper - range % strip;
for (int i=lower, split, strip) {
 for (int j=0, strip) {
  b[i+j] = a[(i+j)-1]
         + a[i+j]
         + a[(i+j)+1];
 }
}
for (int i = split, upper) {
 b[i] = a[i-1]
      + a[i]
      + a[i+1];
}
```

Meaning-preserving !!

Optimisation: Strip Mining

Motivation:

- Strip mining by itself brings no advantage !
- Strip mining triggers loop unrolling !!
- Loop unrolling triggers a plethora of further optmisations !!!

Optimisation: Strip Mining

Motivation:

- Strip mining by itself brings no advantage !
- Strip mining triggers loop unrolling !!
- Loop unrolling triggers a plethora of further optmisations !!!

Example:

```
for (int i = lower, split, 4) {
 for (int j = 0, 4) {
  b[i+j] = a[(i+j)-1] + a[i+j] + a[(i+j)+1];
 }
}
```

```
for (int i = lower, split, 4) {
 b[i+0] = a[(i+0)-1] + a[i+0] + a[(i+0)+1];
 b[i+1] = a[(i+1)-1] + a[i+1] + a[(i+1)+1];
 b[i+2] = a[(i+2)-1] + a[i+2] + a[(i+2)+1];
 b[i+3] = a[(i+3)-1] + a[i+3] + a[(i+3)+1];
}
```

A Taxonomy of Optimisations

Local Optimisation:

- Scope of optimisation restricted to single basic block
- Code transformation focuses on expression evaluation
- Examples: all optimisations so far

Intra-procedural optimisation:

- Scope of optimisation restricted to individual procedure
- Prevailing motivation: improve control flow
- Examples: loop optimisations

Inter-procedural optimisation:

- Whole program view
- In the focus: procedure calls
- Examples: function inlining, specialisation

Optimisation: Function Inlining

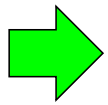
Characteristics:

- Replace function call with instantiated body of function definition
- Integrate local variables of inline function into context function
- Add prologue and epilogue code to map local variables of context function to parameters of inline function and results back

Optimisation: Function Inlining

Example:

```
int inc( int a)
{
  int b;
  b = a + 1;
  return b;
}

int foo( int x)
{
  int y;
  ...
  y = inc( x);
  ...
}
```



```
int foo( int x)
{
  int y;
  int __inc_a,
  int __inc_b;
  ...

  __inc_a = x;
  __inc_b = __inc_a + 1;
  y = __inc_b;
  ...
}
```

Optimisation: Function Inlining

Motivation:

- Avoid function call overhead at runtime
- Create larger optimisation context
- Specialise inline function for individual call site

Optimisation: Function Inlining

Motivation:

- Avoid function call overhead at runtime
- Create larger optimisation context
- Specialise inline function for individual call site

Trade-off:

- Increased code size ($\longrightarrow$ instruction cache)

Optimisation: Function Specialisation

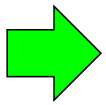
Characteristics:

- Duplicate function definition to exploit context information available at certain call site(s).
- Example: Some arguments are constants.
- Call different version of one original function at different call sites

Optimisation: Function Specialisation

Example:

```
int inc( int a, int val)
{
  int b;
  b = a + val;
  return b;
}

int foo( int x)
{
  int y;
  ...
  y = inc( x, 1);
  ...
}
```



```
int inc__1( int a)
{
  int b;
  b = a + 1;
  return b;
}

int foo( int x)
{
  int y;
  ...
  y = inc__1( x);
  ...
}
```

Optimisation: Function Specialisation

Motivation:

- Create larger (virtual) optimisation context
- Several call sites may still share the same specialisation
  ($\longrightarrow$ less code size increase than with inlining)

Optimisation: Function Specialisation

Motivation:

- Create larger (virtual) optimisation context
- Several call sites may still share the same specialisation ($\longrightarrow$ less code size increase than with inlining)

Trade-off:

- Increased code size ($\longrightarrow$ instruction cache)

A Taxonomy of Optimisations

Local Optimisation:

- Scope of optimisation restricted to single basic block
- Code transformation focuses on expression evaluation
- Examples: all optimisations so far

Intra-procedural optimisation:

- Scope of optimisation restricted to individual procedure
- Prevailing motivation: improve control flow
- Examples: loop optimisations

Inter-procedural optimisation:

- Whole program view
- In the focus: procedure calls
- Examples: function inlining, specialisation

How can we formally describe code transformations ?

Compilation Scheme:

$$\mathcal{C}[\![...]\!] \Rightarrow \quad ... \quad \Big| \, \mathit{guard}$$

How can we formally describe code transformations ?

Compilation Scheme:

$$\mathcal{C}\,[\![...]\!] \Rightarrow \quad ... \;\Big|\, \mathit{guard}$$

Characteristics:

- Scheme has unique name
- Scheme has parameters
- First parameter is pattern matching on code
  - normal font: literal text
  - italics font: placeholder variable
- Further parameters can be anything: numbers, sets, lists, ...
- Scheme defines alternative mappings of pattern into code
- Selection of mapping by top-down evaluation of guards
- Result contains literal text (normal font) and placeholder variables from the pattern (italics)
- Scheme may use various auxiliary schemes

Compilation Schemes

Example: loop unrolling:

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\text{for (int } i{=}lower,upper)\ \{\\ \quad Body\\ \}\\ Rest\end{array}\right]\!\!\right]$$

$$\Rightarrow \left|\begin{array}{l}\{\ \text{int } i = lower;\\ \quad \mathcal{CX}\,[\![Body\ \ i{=}i{+}1;]\!]\,[\![lower]\!]\,[\![upper]\!]\\ \}\\ \mathcal{C}\,[\![Rest]\!]\end{array}\right. \quad upper - lower \leq threshold$$

$$\Rightarrow \left|\begin{array}{l}\text{for (int } i{=}lower,upper)\ \{\\ \quad Body\\ \}\\ \mathcal{C}\,[\![Rest]\!]\end{array}\right. \quad otherwise$$

Compilation Schemes

Example: loop unrolling (continued):

$$\mathcal{CX}\,[\![Code]\!]\,[\![current]\!]\,[\![upper]\!]$$

$$\Rightarrow \quad \begin{array}{l} Code \\ \mathcal{CX}\,[\![Code]\!]\,[\![current+1]\!]\,[\![upper]\!] \end{array} \;\Big|\; current < upper - 1$$

$$\Rightarrow \quad Code \;\Big|\; otherwise$$

Compilation Schemes

Do you spot the errors

in the compilation scheme ?

Compilation Schemes

Example: loop unrolling (corrected):

$$\mathcal{C}\left[\!\!\left[\begin{array}{l}\text{for (int } i{=}lower,upper)\ \{\\ \quad Body\\ \}\\ Rest\end{array}\right]\!\!\right]$$

$$\Rightarrow \left|\begin{array}{l}\{\ \text{int } i = lower;\\ \quad \mathcal{CX}\,[\![\mathcal{C}\,[\![Body]\!]\ \ i{=}i{+}1;]\!]\,[\![lower]\!]\,[\![upper]\!]\\ \}\\ \mathcal{C}\,[\![Rest]\!]\end{array}\right. \quad \begin{array}{l} upper - lower\\ \quad \leq threshold\end{array}$$

$$\Rightarrow \left|\begin{array}{l}\text{for (int } i{=}lower,upper)\ \{\\ \quad \mathcal{C}\,[\![Body]\!]\\ \}\\ \mathcal{C}\,[\![Rest]\!]\end{array}\right. \quad otherwise$$

Interplay of Optimisations

Design principle:

- Keep optimisations orthogonal !
- Do not re-invent / re-implement the wheel (lots of wheels)
- Orchestrate the systematic interplay of optimisations

Interplay of Optimisations

Design principle:

- Keep optimisations orthogonal !
- Do not re-invent / re-implement the wheel (lots of wheels)
- Orchestrate the systematic interplay of optimisations

Optimisation fixed point interation:

- Run optimisations in a cycle
- Stop when no optimisation successful

Interplay of Optimisations

Design principle:

- Keep optimisations orthogonal !
- Do not re-invent / re-implement the wheel (lots of wheels)
- Orchestrate the systematic interplay of optimisations

Optimisation fixed point interation:

- Run optimisations in a cycle
- Stop when no optimisation successful

Caution:

- Ping-pong optimisations
- Make sure there is indeed a fixed point !

Optimisation Interplay Example

Initial code:

```
for (int j = 1, n) {
 for (int i = 0, m) {
  a[i,j] = b[i,j] + b[i,j-1];
 }
}
```

Optimisation Interplay Example

Initial code:

```
for (int j = 1, n) {
 for (int i = 0, m) {
  a[i,j] = b[i,j] + b[i,j-1];
 }
}
```

How do we start ?

Optimisation Interplay Example

Step 1: Loop interchange:

```
for (int j = 1, n) {
 for (int i = 0, m) {
  a[i,j] = b[i,j] + b[i,j-1];
 }
}
```

```
for (int i = 0, m) {
 for (int j = 1, n) {
  a[i,j] = b[i,j] + b[i,j-1];
 }
}
```

Optimisation Interplay Example

Step 2: Strip mining:

```
for (int i = 0, m) {
 for (int j = 1, n) {
  a[i,j] = b[i,j] + b[i,j-1];
}}
```

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  for (int k = 0, 3) {
   a[i,j+k] = b[i,j+k] + c[i,(j+k)-1];
}}}
```

Simplified version, assuming 3 divides n-1 !!

Optimisation Interplay Example

Step 3: Common subexpresssion elimination:

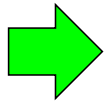
```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  for (int k = 0, 3) {
   a[i,j+k] = b[i,j+k] + b[i,(j+k)-1];
}}}
```

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  for (int k = 0, 3) {
   t = j+k;
   a[i,t] = b[i,t] + b[i,t-1];
}}}
```

Optimisation Interplay Example

Step 4: Loop unrolling:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  for (int k = 0, 3) {
   t = j+k;
   a[i,t] = b[i,t]
            + b[i,t-1];
  }
 }
}
```

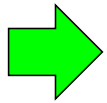


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Optimisation Interplay Example

Step 5: Constant propagation:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

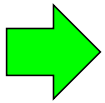


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j + 0;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 0 + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Optimisation Interplay Example

Step 5: Constant propagation:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```



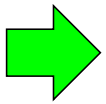
```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j + 0;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 0 + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Important: original code is not removed !!

Optimisation Interplay Example

Step 6: Algebraic simplification:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j + 0;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 0 + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

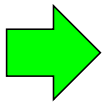


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Optimisation Interplay Example

Step 7: Variable and constant propagation:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
  k = k + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

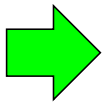


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j;
  a[i,j] = b[i,j] + b[i,j-1];
  k = 1;
  t = j + 1;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 1 + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Optimisation Interplay Example

Step 8: Constant folding and propagation:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j;
  a[i,j] = b[i,j] + b[i,j-1];
  k = 1;
  t = j + 1;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 1 + 1;
  t = j + k;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

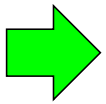


```
for (int i = 0, m) {
 for (int j = 0, n, 3) {
  k = 0;
  t = j;
  a[i,j] = b[i,j] + b[i,j-1];
  k = 1;
  t = j + 1;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 2;
  t = j + 2;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Optimisation Interplay Example

Step 9: Dead code elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  k = 0;
  t = j;
  a[i,j] = b[i,j] + b[i,j-1];
  k = 1;
  t = j + 1;
  a[i,t] = b[i,t] + b[i,t-1];
  k = 2;
  t = j + 2;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

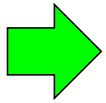


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  a[i,j] = b[i,j] + b[i,j-1];
  t = j + 1;
  a[i,t] = b[i,t] + b[i,t-1];
  t = j + 2;
  a[i,t] = b[i,t] + b[i,t-1];
 }
}
```

Optimisation Interplay Example

Step 10: Array dimension reduction:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  a[i,j] = b[i,j]
         + b[i,j-1];
  t = j + 1;
  a[i,t] = b[i,t]
         + b[i,t-1];
  t = j + 2;
  a[i,t] = b[i,t]
         + b[i,t-1];
 }
}
```

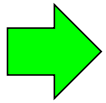


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  a[i*n+j] = b[i*n+j]
           + b[i*n+(j-1)];
  t = j + 1;
  a[i*n+t] = b[i*n+t]
           + b[i*n+(t-1)];
  t = j + 2;
  a[i*n+t] = b[i*n+t]
           + b[i*n+(t-1)];
 }
}
```

Optimisation Interplay Example

Step 11: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  a[i*n+j] = b[i*n+j]
           + b[i*n+(j-1)];
  t = j + 1;
  a[i*n+t] = b[i*n+t]
           + b[i*n+(t-1)];
  t = j + 2;
  a[i*n+t] = b[i*n+t]
           + b[i*n+(t-1)];
 }
}
```

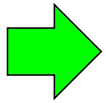


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  a[p+j] = b[p+j]
         + b[p+(j-1)];
  t = j + 1;
  a[p+t] = b[p+t]
         + b[p+(t-1)];
  t = j + 2;
  a[p+t] = b[p+t]
         + b[p+(t-1)];
 }
}
```

Optimisation Interplay Example

Step 12: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  a[p+j] = b[p+j]
           + b[p+(j-1)];
  t = j + 1;
  a[p+t] = b[p+t]
           + b[p+(t-1)];
  t = j + 2;
  a[p+t] = b[p+t]
           + b[p+(t-1)];
 }
}
```

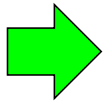


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  a[q] = b[q] + b[p+(j-1)];
  t = j + 1;
  r = p + t;
  a[r] = b[r] + b[p+(t-1)];
  t = j + 2;
  s = p + t;
  a[s] = b[s] + b[p+(t-1)];
 }
}
```

Optimisation Interplay Example

Step 13: Extended algebraic simplification:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  a[q] = b[q] + b[p+(j-1)];
  t = j + 1;
  r = p + t; // p + (j + 1)
  a[r] = b[r] + b[p+(t-1)];
  t = j + 2;
  s = p + t; // p + (j + 2)
  a[s] = b[s] + b[p+(t-1)];
 }
}
```

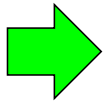


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  a[q] = b[q] + b[(p+j)-1];
  t = j + 1;
  r = (p + j) + 1;
  a[r] = b[r] + b[(p+t)-1];
  t = j + 2;
  s = (p + j) + 2;
  a[s] = b[s] + b[(p+t)-1];
 }
}
```

Optimisation Interplay Example

Step 14: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  a[q] = b[q] + b[(p+j)-1];
  t = j + 1;
  r = (p + j) + 1;
  a[r] = b[r] + b[(p+t)-1];
  t = j + 2;
  s = (p + j) + 2;
  a[s] = b[s] + b[(p+t)-1];
 }
}
```

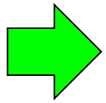


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  a[q] = b[q] + b[q-1];
  t = j + 1;
  r = q + 1;
  a[r] = b[r] + b[(p+t)-1];
  t = j + 2;
  s = q + 2;
  a[s] = b[s] + b[(p+t)-1];
 }
}
```

Optimisation Interplay Example

Step 15: Expression lifting (or flattening):

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  a[q] = b[q] + b[q-1];
  t = j + 1;
  r = q + 1;
  a[r] = b[r] + b[(p+t)-1];
  t = j + 2;
  s = q + 2;
  a[s] = b[s] + b[(p+t)-1];
 }
}
```

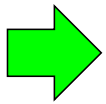


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  t = j + 1;
  r = q + 1;
  v = (p + t) - 1;
  a[r] = b[r] + b[v];
  t = j + 2;
  s = q + 2;
  w = (p + t) - 1;
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 16: Extended algebraic simplification:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  t = j + 1;
  r = q + 1;
  v = (p + t) - 1; // (p+(j+1))-1
  a[r] = b[r] + b[v];
  t = j + 2;
  s = q + 2;
  w = (p + t) - 1; // (p+(j+2))-1
  a[s] = b[s] + b[w];
 }
}
```

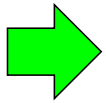


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  t = j + 1;
  r = q + 1;
  v = (p+j) + (1-1);
  a[r] = b[r] + b[v];
  t = j + 2;
  s = q + 2;
  w = (p+j) + (2-1);
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 17: Dead code elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  t = j + 1;
  r = q + 1;
  v = (p+j) + (1-1);
  a[r] = b[r] + b[v];
  t = j + 2;
  s = q + 2;
  w = (p+j) + (2-1);
  a[s] = b[s] + b[w];
 }
}
```

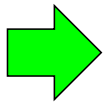


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = (p+j) + (1-1);
  a[r] = b[r] + b[v];
  s = q + 2;
  w = (p+j) + (2-1);
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 18: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = (p+j) + (1-1);
  a[r] = b[r] + b[v];
  s = q + 2;
  w = (p+j) + (2-1);
  a[s] = b[s] + b[w];
 }
}
```

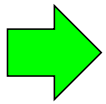


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q + (1-1);
  a[r] = b[r] + b[v];
  s = q + 2;
  w = q + (2-1);
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 19: Constant folding:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q + (1-1);
  a[r] = b[r] + b[v];
  s = q + 2;
  w = q + (2-1);
  a[s] = b[s] + b[w];
 }
}
```

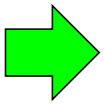


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q + 0;
  a[r] = b[r] + b[v];
  s = q + 2;
  w = q + 1;
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 20: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q + 0;
  a[r] = b[r] + b[v];
  s = q + 2;
  w = q + 1;
  a[s] = b[s] + b[w];
 }
}
```

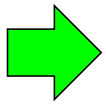


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q + 0;
  a[r] = b[r] + b[v];
  s = q + 2;
  w = r;
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 21: Algebraic simplification:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q + 0;
  a[r] = b[r] + b[v];
  s = q + 2;
  w = r;
  a[s] = b[s] + b[w];
 }
}
```

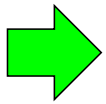


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q;
  a[r] = b[r] + b[v];
  s = q + 2;
  w = r;
  a[s] = b[s] + b[w];
 }
}
```

Optimisation Interplay Example

Step 22: Variable propagation:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q;
  a[r] = b[r] + b[v];
  s = q + 2;
  w = r;
  a[s] = b[s] + b[w];
 }
}
```

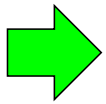


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q;
  a[r] = b[r] + b[q];
  s = q + 2;
  w = r;
  a[s] = b[s] + b[r];
 }
}
```

Optimisation Interplay Example

Step 23: Dead code elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  v = q;
  a[r] = b[r] + b[q];
  s = q + 2;
  w = r;
  a[s] = b[s] + b[r];
 }
}
```

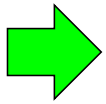


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  a[r] = b[r] + b[q];
  s = q + 2;
  a[s] = b[s] + b[r];
 }
}
```

Optimisation Interplay Example

Step 24: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  a[r] = b[r] + b[q];
  s = q + 2;
  a[s] = b[s] + b[r];
 }
}
```

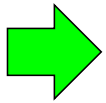


```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```

Optimisation Interplay Example

Step 24: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  a[r] = b[r] + b[q];
  s = q + 2;
  a[s] = b[s] + b[r];
 }
}
```



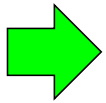
```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```

But is this step really correct ??

Optimisation Interplay Example

Step 24: Common subexpression elimination:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  a[q] = b[q] + b[u];
  r = q + 1;
  a[r] = b[r] + b[q];
  s = q + 2;
  a[s] = b[s] + b[r];
 }
}
```



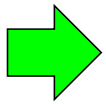
```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```

Arrays a and b must be disjoint !!

Optimisation Interplay Example

Step 25: Loop invariant code motion:

```
for (int i = 0, m) {
 for (int j = 1, n, 3) {
  p = i * n;
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```

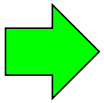


```
for (int i = 0, m) {
 p = i * n;
 for (int j = 1, n, 3) {
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```

Optimisation Interplay Example

Step 26:

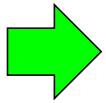
```
for (int i = 0, m) {
 p = i * n;
 for (int j = 1, n, 3) {
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```



Optimisation Interplay Example

Step 26:

```
for (int i = 0, m) {
 p = i * n;
 for (int j = 1, n, 3) {
  q = p + j;
  u = q - 1;
  x = b[q];
  a[q] = x + b[u];
  r = q + 1;
  y = b[r];
  a[r] = y + x;
  s = q + 2;
  a[s] = b[s] + y;
 }
}
```



Done !!

Optimisation Sequence: Research Questions

What is the optimal order of optimisations ?

- Code expanding vs code shrinking transformations
- Some optimisations trigger each other
- Optmisation speed / overhead

Optimisation Sequence: Research Questions

What is the optimal order of optimisations ?

- Code expanding vs code shrinking transformations
- Some optimisations trigger each other
- Optmisation speed / overhead

Do we always find a fixed point ?

- May we get trapped in transformation cycles ?
- Some transformation undoes the effect of another

Optimisation Sequence: Research Questions

What is the optimal order of optimisations ?

- Code expanding vs code shrinking transformations
- Some optimisations trigger each other
- Optmisation speed / overhead

Do we always find a fixed point ?

- May we get trapped in transformation cycles ?
- Some transformation undoes the effect of another

Do we always find the same fixed point ?

- Does one optimisation preclude others ?
- Can we make wrong strategic choices ?

Optimisation Sequence: Research Questions

What is the optimal order of optimisations ?

- Code expanding vs code shrinking transformations
- Some optimisations trigger each other
- Optmisation speed / overhead

Do we always find a fixed point ?

- May we get trapped in transformation cycles ?
- Some transformation undoes the effect of another

Do we always find the same fixed point ?

- Does one optimisation preclude others ?
- Can we make wrong strategic choices ?

What is the impact of heuristic control ?

- Maximum trip count for unrolling
- Maximum code size for inlining
- ...

Optimisation Infrastructure

Problem:

- All non-trivial optimisations require data-flow analysis

Example:

```
n = 4;
...
for (int i = 0, n) {
  ...
}
```

Issues:

- Can the loop be unrolled ?
- Depends on code in "..."
- This "..." can be quite complicated !
- Are there aliases for n ?
- ...

Optimisation Infrastructure

Another problem: anti-dependencies:

```
...
a = foo( b, c);
b = bar( b, c);
...
```

Issues:

- Assume `foo` and `bar` are pure functions without side-effects
- The order of `foo` and `bar` cannot be changed
- Although none of the results is used in the other expression

Optimisation Infrastructure

Laymen's solution:

- Each optimisation implements a tailor-made data-flow analysis

Optimisation Infrastructure

Laymen's solution:

- Each optimisation implements a tailor-made data-flow analysis

Expert's solution:

- Convert internal representation of code into a more optimisation-friendly format

Optimisation Infrastructure

Laymen's solution:

- Each optimisation implements a tailor-made data-flow analysis

Expert's solution:

- Convert internal representation of code into a more optimisation-friendly format

Requirements:

- Relate data flow to control flow in a natural way

Optimisation Infrastructure

Laymen's solution:

- Each optimisation implements a tailor-made data-flow analysis

Expert's solution:

- Convert internal representation of code into a more optimisation-friendly format

Requirements:

- Relate data flow to control flow in a natural way

Static Single Assignment Form (SSA):

- Developed at IBM in the 1980s
- Credits: Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck

Static Single Assignment Form (SSA)

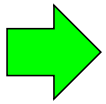
Central idea:

- Each variable has a unique point of definition
- Every applied occurrence can directly be linked to definition
- (Similar to having a link to the point of declaration)
- The latter gives you static information about type, name, etc
- The former gives you information on value (at least abstract)

Static Single Assignment Form (SSA)

Example: linear control flow:

```
a = 10;
b = a + 1;
c = a * b;
a = b + c;
c = c + 1;
a = a + c;
```



```
a_0 = 10;
b_0 = a_0 + 1;
c_0 = a_0 * b_0;
a_1 = b_0 + c_0;
c_1 = c_0 + 1;
a_2 = a_1 + c_1
```

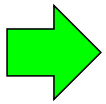
Idea:

- Introduce a counter for each variable
- Enumerate different instances of each variable
- Explicate scoping rules
- Eliminate anti-dependencies

Static Single Assignment Form (SSA)

Example: conditional branches:

```
a = 10;
b = a + 1;
if (a < b) {
 a = b + a;
 c = a + 1;
}
else {
 a = b - a;
 c = a - 1;
}
a = a + b + c;
```

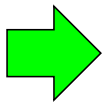


How can we establish SSA form here ?

Static Single Assignment Form (SSA)

Example: conditional branches:

```
a = 10;
b = a + 1;
if (a < b) {
 a = b + a;
 c = a + 1;
}
else {
 a = b - a;
 c = a - 1;
}
a = a + b + c;
```



```
a_0 = 10;
b_0 = a_0 + 1;
if (a_0 < b_0) {
 a_1 = b_0 + a_0;
 c_0 = a_1 + 1;
}
else {
 a_2 = b_0 - a_0;
 c_1 = a_2 - 1;
}
a_3 = phi(a_1,a_2);
c_2 = phi(c_0,c_1);
a_4 = a_3 + b_0 + c_2;
```

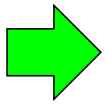
Solution:

- Introduce pseudo functions (phi)
- Explicitly join data flow at control flow join points

Static Single Assignment Form (SSA)

Example: loops:

```
a = 10;
b = a + 1;
while (b > 0) {
 c = b * a;
 b = b - 1;
 a = c;
}
a = a + b;
```



```
a = 10;
b = a + 1;
p = b > 0;
while (p) {
 c = b * a;
 b = b - 1;
 a = c;
 p = b > 0;
}
a = a + b;
```

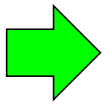
Solution:

- First, lift the predicate expression:
  - one copy in front of the loop
  - one copy at the end of the loop body

Static Single Assignment Form (SSA)

Example: loops:

```
a = 10;
b = a + 1;
p = b > 0;
while (p) {
 c = b * a;
 b = b - 1;
 a = c;
 p = b > 0;
}
a = a + b;
```



```
a_0 = 10;
b_0 = a_0 + 1;
p_0 = b_0 > 0;
while (phi(p_0, p_1)) {
 b_1 = phi(b_0, b_2);
 a_1 = phi(a_0, a_2);
 c_0 = b_1 * a_1;
 b_2 = b_1 - 1;
 a_2 = c_0;
 p_1 = b_2 > 0;
}
a_3 = phi(a_0, a_2);
b_3 = phi(b_0, b_2);
a_4 = a_3 + b_3;
```

Solution:

- Second, introduce more phi pseudo functions

Static Single Assignment Form (SSA)

Creating SSA form:

- Non-trivial code transformation
- Requires accurate data flow analysis
- Compilers must be conservative

Static Single Assignment Form (SSA)

Creating SSA form:

- Non-trivial code transformation
- Requires accurate data flow analysis
- Compilers must be conservative

Removing SSA form:

- Problem: Machines do not have phi function
- Removal algorithm again non-trivial
- Reason: Optimisation may have removed simple properties that still hold after introduction of SSA form

Static Single Assignment Form (SSA)

Conclusion:

- SSA form is a powerful IR that combines
  - control flow aspects and
  - data flow aspects
- SSA form is readable even in textual form
- Variables have single point of assignment
- The associated expression contains all information available on the value
- Code restructuring flexible: absence of anti-dependencies
- SSA form is used in many serious compiler projects
- Examples: most modern compilers, including gcc

Conclusion: Opportunities for Optimisation

Reducing the overhead of abstraction:

- Restructure control flow constructs
- Remove function abstractions

Conclusion: Opportunities for Optimisation

Reducing the overhead of abstraction:

- Restructure control flow constructs
- Remove function abstractions

Taking advantage of special cases:

- Function inlining
- Function specialisation
- Constant folding

Conclusion: Opportunities for Optimisation

Reducing the overhead of abstraction:

- Restructure control flow constructs
- Remove function abstractions

Taking advantage of special cases:

- Function inlining
- Function specialisation
- Constant folding

Adapt code to execution environment properties:

- Strength reduction
- Cache memory utilisation considerations

Conclusion: Considerations for Optimisation

Safety:

- Is the transformation really meaning-preserving ?
- What about odd cases ?
- What about termination behaviour ?
- What about runtime errors ?

Conclusion: Considerations for Optimisation

Profitability:

- Is the effort of optimisation worthwhile ?
- Man-millenia have been wasted developing “useless” optimisations !
- What is impact of one transformation ?
- How often can the transformation be applied ?

Conclusion: Considerations for Optimisation

Profitability:

- Is the effort of optimisation worthwhile ?
- Man-millenia have been wasted developing "useless" optimisations !
- What is impact of one transformation ?
- How often can the transformation be applied ?

Popular principle::

- Make the frequent case fast
- Make the fast case frequent

Conclusion: Considerations for Optimisation

Risk:

- Does the optimisation have side-effects ? Trade-offs ?
- When do the trade-offs adversely affect runtime behaviour ?
- Can adverse effects be ruled out ?
- Will the trade-off between positive and negative effects change in the future ?

The End: Questions ?