Regional Optimizations

Martin Kellogg

Course Announcements

- Graded midterms are at the front of the room
  - If you don't have it yet, pick it up after class
  - If you take it with you, I won't accept regrade requests
- A problem with the PA3c3 autograder was found over the weekend
  - I've therefore granted an extension to today (AoE)
    - Same extension for PA3
- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.

Agenda

- Finish survey of local optimizations
- Deep dive on how to implement local value numbering
- Intro to regional optimizations
- Lifting local value numbering to a regional optimization
- Other regional optimizations

Agenda

- **Finish survey of local optimizations**
- Deep dive on how to implement local value numbering
- Intro to regional optimizations
- Lifting local value numbering to a regional optimization
- Other regional optimizations

Review: Local Optimizations

Review: Local Optimizations

- **Algebraic Simplification** uses math to rewrite expressions
  - e.g., `x + 0` -> `x`

Review: Local Optimizations

- **Algebraic Simplification** uses math to rewrite expressions
  - e.g., `x + 0` -> `x`
- **Constant Folding** computes values at compile time
  - e.g., `5 + 3` -> `8`

Review: Local Optimizations

- **Algebraic Simplification** uses math to rewrite expressions
  - e.g., `x + 0` -> `x`
- **Constant Folding** computes values at compile time
  - e.g., `5 + 3` -> `8`
- **Single Static Assignment** (**SSA**) form is a useful IR for optimizations, because all variables are referentially transparent

Review: Local Optimizations

- Algebraic Simplification uses math to rewrite expressions
  - e.g., `x + 0` -> `x`
- Constant Folding computes values at compile time
  - e.g., `5 + 3` -> `8`
- Single Static Assignment (SSA) form is a useful IR for optimizations, because all variables are referentially transparent
- Common Subexpression Elimination replaces duplicated right-hand sides of expressions
  - e.g., if `x := y + z` and `w := y + z`, then `w := x`

Review: Local Optimizations

- **Algebraic Simplification** uses math to rewrite expressions
  - e.g., `x + 0` -> `x`
- **Constant Folding** computes values at compile time
  - e.g., `5 + 3` -> `8`
- **Single Static Assignment** (**SSA**) form is a useful IR for optimizations, because all variables are referentially transparent
- **Common Subexpression Elimination** replaces duplicated right-hand sides of expressions
  - e.g., if `x := y + z` and `w := y + z`, then `w := x`
- **Copy Propagation** replaces the LHS of assignments with the RHS
  - e.g., if `x := y`, replace subsequent uses of `x` with `y`

Dead Code Elimination (DCE)

Dead Code Elimination (DCE)

- If:
    - `w := rhs` appears in a basic block
    - `w` does not appear anywhere else in the program

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result
- Example (assume that a is not used anywhere else)

```
x := z + y
a := x          ⟶
x := 2 * a
```

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result
- Example (assume that `a` is not used anywhere else)

```
x := z + y          b := z + y
a := x      →       a := b      →
x := 2 * a          x := 2 * b
```

Dead Code Elimination (DCE)

- If:
  - `w := rhs` appears in a basic block
  - `w` does not appear anywhere else in the program
- Then
  - the statement `w := rhs` is **dead** and can be eliminated
- ***Dead*** = does not contribute to the program's result
- Example (assume that `a` is not used anywhere else)

```
x := z + y          b := z + y          b := z + y
a := x       →      a := b       →      x := 2 * b
x := 2 * a          x := 2 * b
```

Applying Local Optimizations

Applying Local Optimizations

- Each local optimization does very little by itself

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations **interact**
    - Performing one optimization enables (or disables!) other optimizations

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations **interact**
  - Performing one optimization enables (or disables!) other optimizations
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - **Phase ordering problem** again: must beware of local minima

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations **interact**
  - Performing one optimization enables (or disables!) other optimizations
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
  - **Phase ordering problem** again: must beware of local minima
- Interpreters and JITs must be fast!
  - The optimizer can also be stopped at any time to limit the compilation time

An Example

- Initial code:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic simplification:

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

An Example

- Algebraic simplification:

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := c * c
e := b + b
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 3 + 3
f := a + d
g := e * f
```

An Example

- Constant folding:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

An Example

- Common subexpression elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

An Example

- Copy propagation:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x
~~b := 3~~
~~c := x~~
~~d := a~~
~~e := 6~~
f := a + a
g := 6 * f
```

An Example

- Dead code elimination:

```
a := x * x
~~b := 3~~
~~c := x~~
~~d := a~~
~~e := 6~~
f := a + a
g := 6 * f
```

Could we get to `g = 12 * a`?

Local Optimization Notes

Local Optimization Notes

- Intermediate code is helpful for many optimizations

Local Optimization Notes

- Intermediate code is helpful for many optimizations
- “Program optimization” is grossly misnamed

Local Optimization Notes

- Intermediate code is helpful for many optimizations
- “Program optimization” is grossly misnamed
  - Code produced by “optimizers” is **not optimal** in any reasonable sense; “**program improvement**” is a better term

Local Optimization Notes

- Intermediate code is helpful for many optimizations
- “Program optimization” is grossly misnamed
    - Code produced by “optimizers” is **not optimal** in any reasonable sense; “**program improvement**” is a better term
- Even “simple” local optimizations may be **unsafe** in some contexts
    - e.g., can we safely constant fold in a block that may divide by zero?

Local Optimization Notes

- Intermediate code is helpful for many optimizations
- "Program optimization" is grossly misnamed
    - Code produced by "optimizers" is **not optimal** in any reasonable sense; "**program improvement**" is a better term
- Even "simple" local optimizations may be **unsafe** in some contexts
    - e.g., can we safely constant fold in a block that may divide by zero?
- Programmers might protest that they don't write code that can be easily improved by such "simple" optimizations

Local Optimization Notes

- Intermediate code is helpful for many optimizations
- "Program optimization" is grossly misnamed
  - Code produced by "optimizers" is **not optimal** in any reasonable sense; "**program improvement**" is a better term
- Even "simple" local optimizations may be **unsafe** in some contexts
  - e.g., can we safely constant fold in a block that may divide by zero?
- Programmers might protest that they don't write code that can be easily improved by such "simple" optimizations
  - But keep in mind that **the compiler** is actually generating most of the code you're optimizing (e.g., array accesses)

Deep Dive: Local Value Numbering

Deep Dive: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block

Deep Dive: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - You can think of this section as a “how-to” for the **conceptual** optimizations that I covered on Monday and earlier today

Deep Dive: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - You can think of this section as a “how-to” for the **conceptual** optimizations that I covered on Monday and earlier today
  - You can implement many of the other optimizations we’ve discussed (e.g., constant folding) using the same core ideas

Deep Dive: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - You can think of this section as a "how-to" for the **conceptual** optimizations that I covered on Monday and earlier today
  - You can implement many of the other optimizations we've discussed (e.g., constant folding) using the same core ideas
- The key idea of LVN is to assign a **distinct number** (called the "***value number***") to **each value** computed by the basic block

Deep Dive: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - You can think of this section as a “how-to” for the **conceptual** optimizations that I covered on Monday and earlier today
  - You can implement many of the other optimizations we’ve discussed (e.g., constant folding) using the same core ideas
- The key idea of LVN is to assign a **distinct number** (called the “***value number***”) to **each value** computed by the basic block
  - LVN’s goal: assign the same number to two different expressions iff they are **provably equal**

LVN: Core Algorithm

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**
  - the LVN hashtable is initially empty

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**
  - the LVN hashtable is initially empty
- For each operation `T := L Op R`, in program order, LVN:

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**
  - the LVN hashtable is initially empty
- For each operation `T := L Op R`, in program order, LVN:
  - looks up `L` and `R` in the hashtable to get VN_L and VN_R

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**
  - the LVN hashtable is initially empty
- For each operation `T := L Op R`, in program order, LVN:
  - looks up `L` and `R` in the hashtable to get VN_L and VN_R
    - if found, use already-assigned value number
    - if not found, assign a new value number

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**
  - the LVN hashtable is initially empty
- For each operation `T := L Op R`, in program order, LVN:
  - looks up `L` and `R` in the hashtable to get VN_L and VN_R
    - if found, use already-assigned value number
    - if not found, assign a new value number
  - creates a new string key k = VN_L `Op` VN_R

LVN: Core Algorithm

- LVN initializes a hashtable for each basic block
  - this table maps **names, constants, and expressions** to **value numbers**
  - the LVN hashtable is initially empty
- For each operation `T := L Op R`, in program order, LVN:
  - looks up `L` and `R` in the hashtable to get VN_L and VN_R
    - if found, use already-assigned value number
    - if not found, assign a new value number
  - creates a new string key `k` = VN_L `Op` VN_R
  - looks up `k` in the table, assigning a new value number if not found

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	0

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	0
c	1

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	0
c	1

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
`b`	0
`c`	1
`0 + 1`	2

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	0
c	1
0 + 1	2
a	2

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
`b`	0
`c`	1
`0 + 1`	2
`a`	2

Note how `a` and `0 + 1` get the **same** value number!

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	0
c	1
0 + 1	2
a	2

LVN: Example

```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

already have an entry for **a**

Value table:

key	value num
b	0
c	1
0 + 1	2
a	2

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
`b`	0
`c`	1
`0 + 1`	2
`a`	2
`d`	3

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	0
c	1
0 + 1	2
a	2
d	3
2 - 3	4

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

Value table:

key	value num
b	~~0~~ 4
c	1
0 + 1	2
a	2
d	3
2 - 3	4

LVN: Example

```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

already have entries for `b, c`

Value table:

key	value num
`b`	~~0~~ 4
`c`	1
`0 + 1`	2
`a`	2
`d`	3
`2 - 3`	4

LVN: Example

```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

Value table:

key	value num
b	~~0~~ 4
c	~~1~~ 5
0 + 1	2
a	2
d	3
2 - 3	4
1 + 4	5

LVN: Example

```
a := b + c
b := a - d
c := b + c
```
$\boxed{\texttt{d := a - d}}$

Value table:

key	value num
b	~~0~~ 4
c	~~1~~ 5
0 + 1	2
a	2
d	3
2 - 3	4
1 + 4	5

LVN: Example

```
a := b + c
b := a - d
c := b + c
```
$\boxed{\texttt{d := a - d}}$

already have an entry for **2 − 3**!

Value table:

key	value num
b	~~0~~ 4
c	~~1~~ 5
0 + 1	2
a	2
d	3
2 - 3	4
1 + 4	5

LVN: Example

```
a  := b + c
b  := a - d
c  := b + c
d  := a - d
```

Value table:

key	value num
b	~~0~~ 4
c	~~1~~ 5
0 + 1	2
a	2
d	~~3~~ 4
2 - 3	4
1 + 4	5

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

How to use this information?

We can replace a - d with anything with value number 4

Value table:

key	value num
b	~~0~~ 4
c	~~1~~ 5
0 + 1	2
a	2
d	~~3~~ 4
2 - 3	4
1 + 4	5

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := a - d
```

How to use this information?
We can replace `a - d` with anything with value number 4!

Value table:

key	value num
`b`	~~0~~ 4
`c`	~~1~~ 5
`0 + 1`	2
`a`	2
`d`	~~3~~ 4
`2 - 3`	4
`1 + 4`	5

LVN: Example

```
a := b + c
b := a - d
c := b + c
d := b
```

Value table:

key	value num
`b`	~~0~~ 4
`c`	~~1~~ 5
`0 + 1`	2
`a`	2
`d`	~~3~~ 4
`2 - 3`	4
`1 + 4`	5

How to use this information?
We can replace `a - d` with anything with value number 4!

LVN: Notes

LVN: Notes

- LVN is **most effective** when the program is already in SSA form
  - Why?

LVN: Notes

- LVN is **most effective** when the program is already in SSA form
  - Why?
    - Avoids the need to overwrite table entries!

LVN: Notes

- LVN is **most effective** when the program is already in SSA form
  - Why?
    - Avoids the need to overwrite table entries!
- It’s easy to build **commutativity** into LVN

LVN: Notes

- LVN is **most effective** when the program is already in SSA form
  - Why?
    - Avoids the need to overwrite table entries!
- It's easy to build **commutativity** into LVN
  - Always put value numbers in order in commutative keys
    - e.g., always `0+1`, never `1+0`

LVN: Notes

- LVN is **most effective** when the program is already in SSA form
  - Why?
    - Avoids the need to overwrite table entries!
- It's easy to build **commutativity** into LVN
  - Always put value numbers in order in commutative keys
    - e.g., always `0+1`, never `1+0`
- LVN can incorporate **other optimizations**
  - e.g., if operands are constants, storing that info directly in the table enables constant folding during LVN

LVN: Notes

- LVN is **most effective** when the program is already in SSA form
  - Why?
    - Avoids the need to overwrite table entries!
- It's easy to build **commutativity** into LVN
  - Always put value numbers in order in commutative keys
    - e.g., always `0+1`, never `1+0`
- LVN can incorporate **other optimizations**
  - e.g., if operands are constants, storing that info directly in the table enables constant folding during LVN
- LVN is highly **order-dependent**: rewriting the code to change the order of operations may change the results

Trivia Break: Literature

This epic 1862 novel follows the lives and interactions of several characters from 1815 until the 1832 June Rebellion in Paris. It is one of the longest novels ever written in French, at 655,478 words. The novel contains many digressions - comprising more than a quarter of its pages - that do not advance the plot in any way. Despite this, Upton Sinclair described it as "one of the half-dozen greatest novels of the world." It has been has been popularized through numerous adaptations for film, television, and the stage, including a musical. Its author is Victor Hugo, whose other works include *The Hunchback of Notre-Dame.*

Trivia Break: Mathematics

This French republican political activist was repeatedly arrested as a teenager in the late 1820s and early 1830s in the lead up to the June Rebellion (which is famously the setting for *Les Misérables*), before dying in a duel just a few days before the uprising began, aged just 20. Despite his involvement in politics, he was an active research mathematician. His work in mathematics, though not appreciated during his lifetime, laid the foundations for two major branches of abstract algebra, one of which is named for him. He also solved a problem open for over 350 years: determining a necessary and sufficient condition for a polynomial to be solvable by radicals.

A Brief Digression on GDB

A Brief Digression on GDB

- It has come to my attention that not all of you have been trained on how to use GDB to debug assembly programs
  - GDB is a tool you *should* all be familiar with…

A Brief Digression on GDB

- It has come to my attention that not all of you have been trained on how to use GDB to debug assembly programs
  - GDB is a tool you *should* all be familiar with…
- I’m now going to do a short demo of how I would approach **debugging a segfault** in an assembly program that I’ve written

A Brief Digression on GDB

- It has come to my attention that not all of you have been trained on how to use GDB to debug assembly programs
  - GDB is a tool you *should* all be familiar with…
- I'm now going to do a short demo of how I would approach **debugging a segfault** in an assembly program that I've written
  - Peanut gallery commentary is **encouraged**: I am by no means the world's best systems programmer

A Brief Digression on GDB

- It has come to my attention that not all of you have been trained on how to use GDB to debug assembly programs
    - GDB is a tool you *should* all be familiar with…
- I'm now going to do a short demo of how I would approach **debugging a segfault** in an assembly program that I've written
    - Peanut gallery commentary is **encouraged**: I am by no means the world's best systems programmer
- The problem: we're generating assembly code for Cool programs that call into libc, and printf is segfaulting
    - You will **not** be able to reproduce this behavior on v1.36 of the reference compiler :)

Regional Optimizations

Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together

Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy

Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target

Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target
- Primary difference between local and regional optimizations is the need to handle **control flow**
  - e.g., if/else, jumps, etc.

Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target
- Primary difference between local and regional optimizations is the need to handle **control flow**
  - e.g., if/else, jumps, etc.
- We will look at two examples:
  - extending local value numbering to regions
  - loop unrolling

Extended Basic Blocks

- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks

Extended Basic Blocks

- Regional optimizations usually work on an ***extended basic block*** (“***EBB***”): a small control-flow graph of basic blocks
- Formally, an EBB is a **maximal** collection of basic blocks where:

Extended Basic Blocks

- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks
- Formally, an EBB is a **maximal** collection of basic blocks where:
    - there are unique entry and exit blocks

Extended Basic Blocks

- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks
- Formally, an EBB is a **maximal** collection of basic blocks where:
  - there are unique entry and exit blocks
  - all basic blocks besides the entry must only have predecessor blocks that are members of the EBB

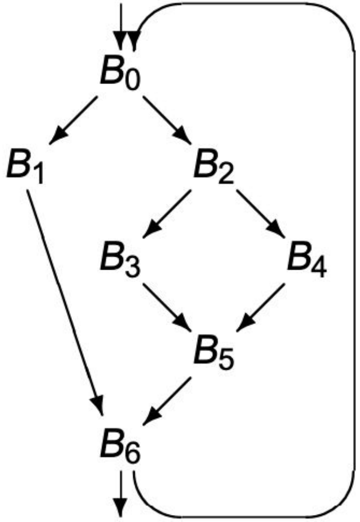
Extended Basic Blocks

- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks
- Formally, an EBB is a **maximal** collection of basic blocks where:
  - there are unique entry and exit blocks
  - all basic blocks besides the entry must only have predecessor blocks that are members of the EBB
- Most local optimizations can operate on EBBs with small modifications (including most of those we saw earlier)
  - Thus, you can do most local optimizations at the regional level!

Extended Basic Blocks

- Regional optimizations usually work on an ***extended basic block*** (“***EBB***”): a small control-flow graph of basic blocks
- Formally, an EBB is a **maximal** collection of basic blocks where:
  - there are unique entry and exit blocks
  - all basic blocks besides the entry must only have predecessor blocks that are members of the EBB
- Most local optimizations can operate on EBBs with small modifications (including most of those we saw earlier)
  - Thus, you can do most local optimizations at the regional level!
- I will show how we extend **local value numbering** to a regional optimization; others are left as an exercise

Extended Basic Blocks: Example



$$
\begin{aligned}
B_0:\ & m_0 \leftarrow a_0 + b_0 \\
& n_0 \leftarrow a_0 + b_0 \\
& (a_0 > b_0) \rightarrow B_1, B_2
\end{aligned}
$$

$$
\begin{aligned}
B_1:\ & p_0 \leftarrow c_0 + d_0 \\
& r_0 \leftarrow c_0 + d_0 \\
& \rightarrow B_6
\end{aligned}
$$

$$
\begin{aligned}
B_2:\ & q_0 \leftarrow a_0 + b_0 \\
& r_1 \leftarrow c_0 + d_0 \\
& (a_0 > b_0) \rightarrow B_3, B_4
\end{aligned}
$$

$$
\begin{aligned}
B_3:\ & e_0 \leftarrow b_0 + 18 \\
& s_0 \leftarrow a_0 + b_0 \\
& u_0 \leftarrow e_0 + f_0 \\
& \rightarrow B_5
\end{aligned}
$$

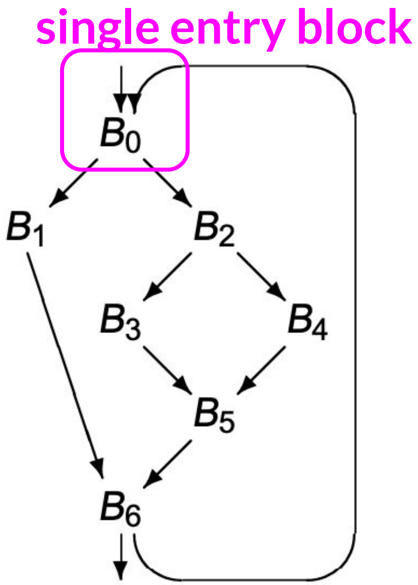
$$
\begin{aligned}
B_4:\ & e_1 \leftarrow a_0 + 17 \\
& t_0 \leftarrow c_0 + d_0 \\
& u_1 \leftarrow e_1 + f_0 \\
& \rightarrow B_5
\end{aligned}
$$

$$
\begin{aligned}
B_5:\ & e_2 \leftarrow \phi(e_0, e_1) \\
& u_2 \leftarrow \phi(u_0, u_1) \\
& v_0 \leftarrow a_0 + b_0 \\
& w_0 \leftarrow c_0 + d_0 \\
& x_0 \leftarrow e_2 + f_0 \\
& \rightarrow B_6
\end{aligned}
$$

$$
\begin{aligned}
B_6:\ & r_2 \leftarrow \phi(r_0, r_1) \\
& y_0 \leftarrow a_0 + b_0 \\
& z_0 \leftarrow c_0 + d_0
\end{aligned}
$$

(don't worry about the details)

Extended Basic Blocks: Example



B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_2$

B_1: $p_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_6$

B_2: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_3, B_4$

B_3: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$
$\rightarrow B_5$

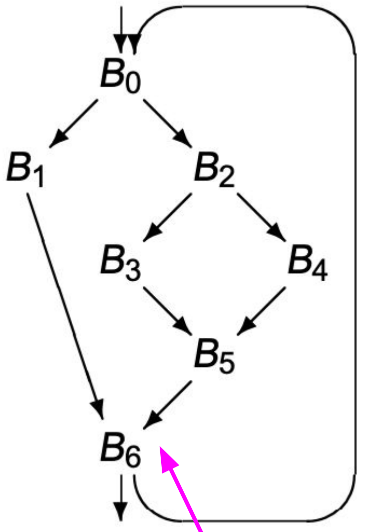
B_4: $e_1 \leftarrow a_0 + 17$
$t_0 \leftarrow c_0 + d_0$
$u_1 \leftarrow e_1 + f_0$
$\rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$u_2 \leftarrow \phi(u_0, u_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow c_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_6$

B_6: $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$

(don't worry about the details)

Extended Basic Blocks: Example



control-flow joins

B_0: $m_0 \leftarrow a_0 + b_0$
$\quad n_0 \leftarrow a_0 + b_0$
$\quad (a_0 > b_0) \rightarrow B_1, B_2$

B_1: $p_0 \leftarrow c_0 + d_0$
$\quad r_0 \leftarrow c_0 + d_0$
$\quad \rightarrow B_6$

B_2: $q_0 \leftarrow a_0 + b_0$
$\quad r_1 \leftarrow c_0 + d_0$
$\quad (a_0 > b_0) \rightarrow B_3, B_4$

B_3: $e_0 \leftarrow b_0 + 18$
$\quad s_0 \leftarrow a_0 + b_0$
$\quad u_0 \leftarrow e_0 + f_0$
$\quad \rightarrow B_5$

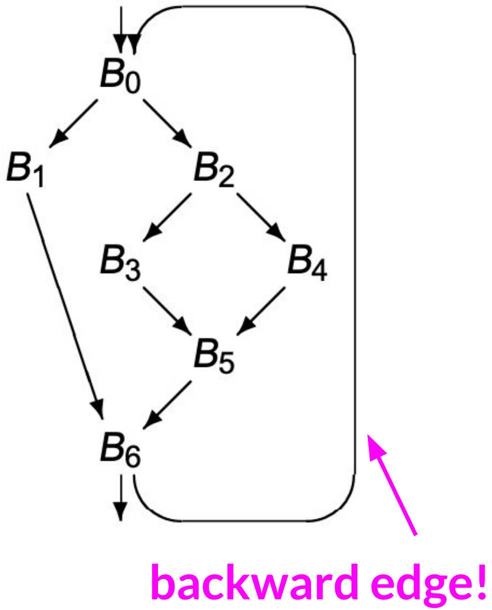
B_4: $e_1 \leftarrow a_0 + 17$
$\quad t_0 \leftarrow c_0 + d_0$
$\quad u_1 \leftarrow e_1 + f_0$
$\quad \rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$\quad u_2 \leftarrow \phi(u_0, u_1)$
$\quad v_0 \leftarrow a_0 + b_0$
$\quad w_0 \leftarrow c_0 + d_0$
$\quad x_0 \leftarrow e_2 + f_0$
$\quad \rightarrow B_6$

B_6: $r_2 \leftarrow \phi(r_0, r_1)$
$\quad y_0 \leftarrow a_0 + b_0$
$\quad z_0 \leftarrow c_0 + d_0$

(don’t worry about the details)

Extended Basic Blocks: Example



B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_2$

B_1: $p_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_6$

B_2: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_3, B_4$

B_3: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$
$\rightarrow B_5$

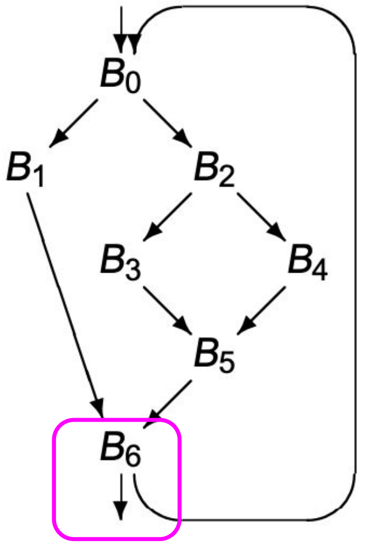
B_4: $e_1 \leftarrow a_0 + 17$
$t_0 \leftarrow c_0 + d_0$
$u_1 \leftarrow e_1 + f_0$
$\rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$u_2 \leftarrow \phi(u_0, u_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow c_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_6$

B_6: $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$

(don’t worry about the details)

Extended Basic Blocks: Example



single exit block

$$
\begin{aligned}
B_0:\;& m_0 \leftarrow a_0 + b_0\\
& n_0 \leftarrow a_0 + b_0\\
& (a_0 > b_0) \rightarrow B_1, B_2
\end{aligned}
$$

$$
\begin{aligned}
B_1:\;& p_0 \leftarrow c_0 + d_0\\
& r_0 \leftarrow c_0 + d_0\\
& \rightarrow B_6
\end{aligned}
$$

$$
\begin{aligned}
B_2:\;& q_0 \leftarrow a_0 + b_0\\
& r_1 \leftarrow c_0 + d_0\\
& (a_0 > b_0) \rightarrow B_3, B_4
\end{aligned}
$$

$$
\begin{aligned}
B_3:\;& e_0 \leftarrow b_0 + 18\\
& s_0 \leftarrow a_0 + b_0\\
& u_0 \leftarrow e_0 + f_0\\
& \rightarrow B_5
\end{aligned}
$$

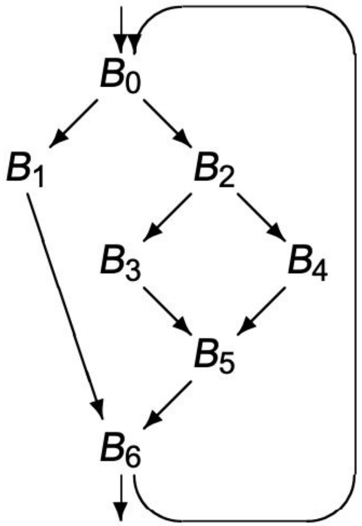
$$
\begin{aligned}
B_4:\;& e_1 \leftarrow a_0 + 17\\
& t_0 \leftarrow c_0 + d_0\\
& u_1 \leftarrow e_1 + f_0\\
& \rightarrow B_5
\end{aligned}
$$

$$
\begin{aligned}
B_5:\;& e_2 \leftarrow \phi(e_0, e_1)\\
& u_2 \leftarrow \phi(u_0, u_1)\\
& v_0 \leftarrow a_0 + b_0\\
& w_0 \leftarrow c_0 + d_0\\
& x_0 \leftarrow e_2 + f_0\\
& \rightarrow B_6
\end{aligned}
$$

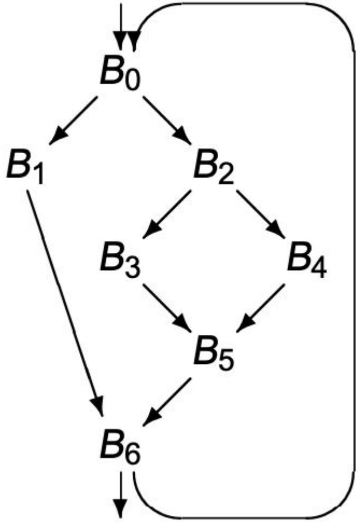
$$
\begin{aligned}
B_6:\;& r_2 \leftarrow \phi(r_0, r_1)\\
& y_0 \leftarrow a_0 + b_0\\
& z_0 \leftarrow c_0 + d_0
\end{aligned}
$$

(don’t worry about the details)

Super-local Value Numbering

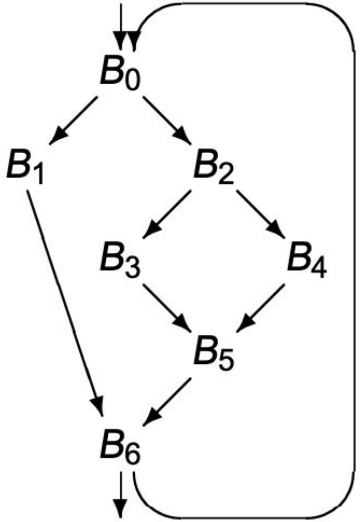


Super-local Value Numbering



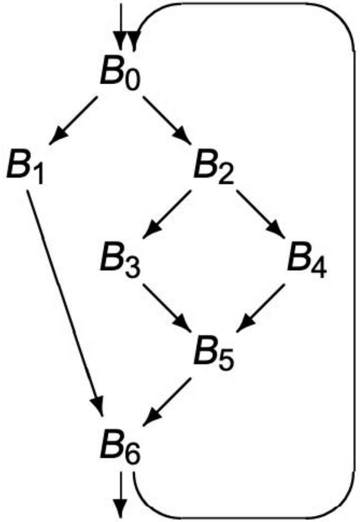
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB

Super-local Value Numbering



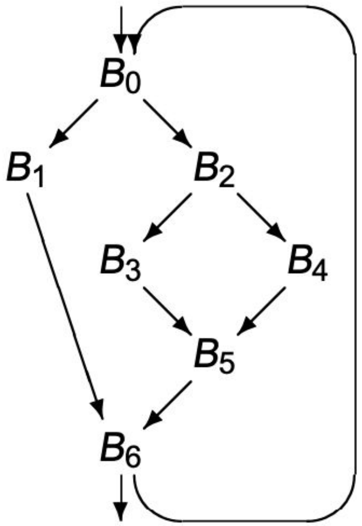
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**

Super-local Value Numbering



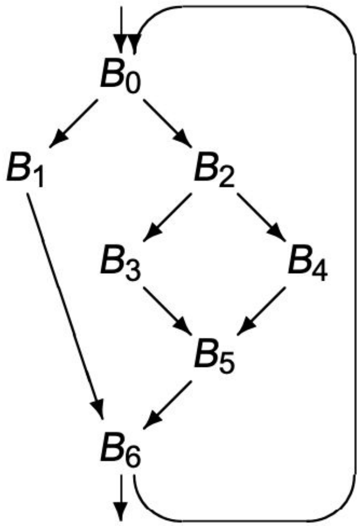
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**
  - That is, treat each path as if it were a block!
    - after all, no branches in a single path…

Super-local Value Numbering



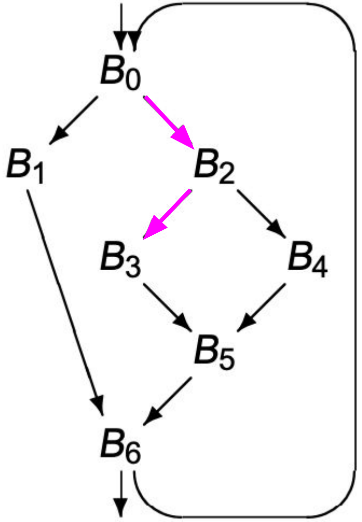
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**
  - That is, treat each path as if it were a block!
    - after all, no branches in a single path...
- Blocks with single predecessor can keep the hashtable from the last block

Super-local Value Numbering



- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**
  - That is, treat each path as if it were a block!
    - after all, no branches in a single path…
- Blocks with single predecessor can keep the hashtable from the last block
- Any block with multiple predecessors, such as B_5, can use a fresh hashtable

S-LVN: Example



consider the path B_0, B_2, B_3

B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_2$

B_1: $p_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_6$

B_2: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_3, B_4$

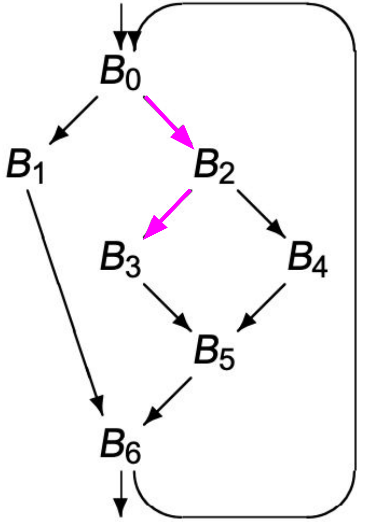
B_3: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$
$\rightarrow B_5$

B_4: $e_1 \leftarrow a_0 + 17$
$t_0 \leftarrow c_0 + d_0$
$u_1 \leftarrow e_1 + f_0$
$\rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$u_2 \leftarrow \phi(u_0, u_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow c_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_6$

B_6: $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$

S-LVN: Example

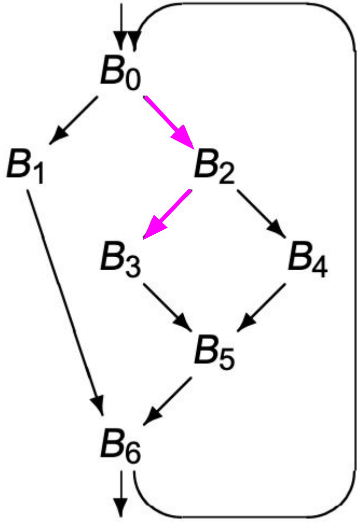


consider the path B_0, B_2, B_3

combine into a single logical block

$$
\begin{array}{lrcl}
B_0: & m_0 & \leftarrow & a_0 + b_0 \\
 & n_0 & \leftarrow & a_0 + b_0 \\
 & q_0 & \leftarrow & a_0 + b_0 \\
 & r_1 & \leftarrow & c_0 + d_0 \\
 & e_0 & \leftarrow & b_0 + 18 \\
 & s_0 & \leftarrow & a_0 + b_0 \\
 & u_0 & \leftarrow & e_0 + f_0
\end{array}
$$

S-LVN: Example



consider the path B_0, B_2, B_3

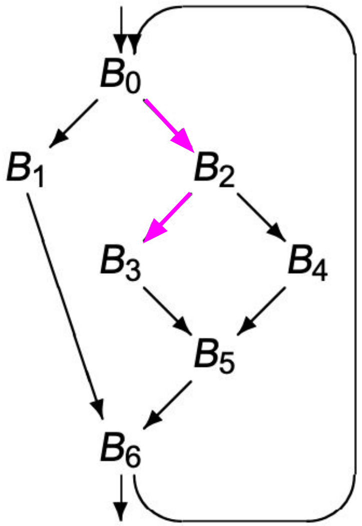
combine into a single logical block

$$
\begin{aligned}
\mathbf{B_0}: m_0 &\leftarrow a_0 + b_0 \\
\boxed{n_0 \leftarrow a_0 + b_0} & \\
q_0 &\leftarrow a_0 + b_0 \\
r_1 &\leftarrow c_0 + d_0 \\
e_0 &\leftarrow b_0 + 18 \\
s_0 &\leftarrow a_0 + b_0 \\
u_0 &\leftarrow e_0 + f_0
\end{aligned}
$$

LVN can discover this redundancy on its own…

S-LVN: Example

combine into a single logical block



consider the path B_0, B_2, B_3

$$
\begin{aligned}
B_0: \; & m_0 \leftarrow a_0 + b_0 \\
& n_0 \leftarrow a_0 + b_0 \\
& \boxed{q_0 \leftarrow a_0 + b_0} \\
& r_1 \leftarrow c_0 + d_0 \\
& e_0 \leftarrow b_0 + 18 \\
& \boxed{s_0 \leftarrow a_0 + b_0} \\
& u_0 \leftarrow e_0 + f_0
\end{aligned}
$$

...but we need S-LVN to discover these two!

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn’t feasible, because paths **grow exponentially** in the number of branches

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn’t feasible, because paths **grow exponentially** in the number of branches
    - this is called the ***path explosion problem***
        - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work
  - insight: paths share **common prefixes**
    - e.g., (B_0, B_2, B_3) and (B_0, B_2, B_4) share prefix (B_0, B_2)

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work
  - insight: paths share **common prefixes**
    - e.g., (B_0, B_2, B_3) and (B_0, B_2, B_4) share prefix (B_0, B_2)
  - the compiler can **cache** the results for common prefixes and reuse them when analyzing related paths

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work
  - insight: paths share **common prefixes**
    - ...d (B_0, B_2, B_4) share prefix (B_0, B_2)
  - ...**ne** the results for common prefixes and ...alyzing related paths

For more details on this algorithm, see the book.

Other Regional Optimizations

- Loop unrolling
- Code motion
- Loop induction variable elimination

Loop Unrolling

Loop Unrolling

- To ***unroll*** a loop, replicate the loop's body and adjust the logic that controls the number of iterations performed

Loop Unrolling

- To ***unroll*** a loop, replicate the loop's body and adjust the logic that controls the number of iterations performed
- Direct benefits:
  - reduce number of branches (they're expensive)
  - enable reuse of certain computations (e.g., outer loop indices)
  - improve spatial locality, especially for array accesses

Loop Unrolling

- To ***unroll*** a loop, replicate the loop's body and adjust the logic that controls the number of iterations performed
- Direct benefits:
    - reduce number of branches (they're expensive)
    - enable reuse of certain computations (e.g., outer loop indices)
    - improve spatial locality, especially for array accesses
- Loop unrolling changes the **ratio** of arithmetic to memory operations in the loop

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
    - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
    - unrolling increases the number of operations in the loop body, which might enable other optimizations
        - e.g., more operations in the loop body might unlock interesting instruction schedules

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
  - unrolling increases the number of operations in the loop body, which might enable other optimizations
    - e.g., more operations in the loop body might unlock interesting instruction schedules
  - unrolling can enable multi-word instructions (i.e., SIMD)
    - ***SIMD*** = "single instruction, multiple data"

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
    - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
    - unrolling increases the number of operations in the loop body, which might enable other optimizations
        - e.g., more operations in the loop body might unlock interesting instruction schedules
    - unrolling can enable multi-word instructions (i.e., SIMD)
        - ***SIMD*** = "single instruction, multiple data"
    - unrolled loop may use more registers, and if it causes a spill the unrolling is almost certainly not worth it

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
  - unrolling increases the number of operations in the loop body, which might enable other optimizations
    - e.g. more operations in the loop body might unlock

uctions (i.e., SIMD)

e data”

and if it causes a spill

the unrolling is almost certainly not worth it

Whether or not to unroll a loop often depends on these factors, so there is **no one-size-fits-all algorithm** for deciding whether to unroll

Code Motion

- Goal: move **loop-invariant calculations** out of loops

Code Motion

- Goal: move **loop-invariant calculations** out of loops
- Example:

```
for (i = 0; i < 10; i++) {
 a[i] = a[i] + b[j];
 z = z + 10000;
}
```

Code Motion

- Goal: move **loop-invariant calculations** out of loops
- Example:

```
for (i = 0; i < 10; i++) {
 a[i] = a[i] + b[j];
 z = z + 10000;
}
```

→

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i++) {
 a[i] = a[i] + t1;
 z = z + t2;
}
```

Code Motion

- Goal: move **loop-invariant calculations** out of loops
- Example:

```
for (i = 0; i < 10; i++) {
 a[i] = a[i] + b[j];
 z = z + 10000;
}
```

→

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i++) {
 a[i] = a[i] + t1;
 z = z + t2;
}
```

- Benefit: avoids **redundant computation** each time around the loop

Loop Induction Variable Elimination

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
    - incremented each time around loop
    - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
    - compute initial offsets/pointers before loop
    - increment offsets/pointers each time around loop

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop
  - then do loop-invariant code motion

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction vari***
    - incremented each time arounc
    - offsets & pointers calculated fr
- If used only to index arrays, rewrite with pointers
    - compute initial offsets/pointers before loop
    - increment offsets/pointers each time around loop
    - no expensive scaling in loop
    - then do loop-invariant code motion

```
for (i = 0; i < 10; i++){
    a[i] = a[i] + x;
}
```

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction vari***
  - incremented each time around
  - offsets & pointers calculated fr
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offset
  - no expensive sca
  - then do loop-inv

```
for (i = 0; i < 10; i++){
    a[i] = a[i] + x;
}
```

```
for (p = &a[0]; p < &a[10]; p = p+4){
    *p = *p + x;
}
```

Summary: Regional Optimizations

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality
  - This means that they are inherently **more risky** than their local cousins

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality
  - This means that they are inherently **more risky** than their local cousins
- Nearly all local optimizations can be extended to work at the regional level

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality
  - This means that they are inherently **more risky** than their local cousins
- Nearly all local optimizations can be extended to work at the regional level
  - Which you want to use is up to you!

Course Announcements

- Graded **midterms** are at the front of the room
  - If you don't have it yet, pick it up after class
  - If you take it with you, I won't accept regrade requests
- A problem with the PA3c3 autograder was found over the weekend
  - I've therefore granted an extension to **today** (AoE)
    - Same extension for PA3
- We recently fixed a bug in the reference compiler's x86-64 module. Only use Cool **version 1.39** for compiling to x86.