Compiler Optimization
Overview of Optimizations

- Goal of optimization is to generate **better** code
  - Impossible to generate **optimal** code
    - Factors beyond control of compiler (user input, OS design, HW design) all affect what is optimal
    - Even discounting above, it’s still an NP-complete problem

- Better one or more of the following (in the average case)
  - Execution time
  - Memory usage
  - Energy consumption
    - To reduce energy bill in a data center
    - To improve the lifetime of battery powered devices
  - Binary Executable Size
    - If binary needs to be sent over the network
    - If binary must fit inside small device with limited storage

- Other criteria

- Should **never** change program semantics
Compiler optimization is essentially a transformation
- Delete / Add / Move / Modify something

Code-related transformations
- Optimizes what code is generated
- Goal: execute least number of least costly instructions

Layout-related transformations
- Optimizes where in memory code and data is placed
- Goal: maximize spatial locality
  - Spatial locality: on an access, likelihood that nearby locations will also be accessed soon
  - Increases likelihood latter accesses will be faster
    E.g. If access fetches cache line, latter accesses can reuse
    E.g. If access results in page fault, latter can reuse page
Layout-Related Optimizations
Two ways to lay out code

```c
f() {
    ... call h();
}
g() {
    ...
}
h() {
    ...
}
```

or

```c
f() {
}

code of h()

g() {
}

code of h()
```

OR

```c
f() {
}

code of h()
```

code of g()
Which Code Layout is Better?

Assume

- data cache has one N-word line
- the size of each function is N/2-word long
- access sequence is “g, f, h, f, h, f, h”

<table>
<thead>
<tr>
<th>Cache</th>
<th>code of f()</th>
<th>code of g()</th>
<th>code of h()</th>
<th>code of f()</th>
<th>code of h()</th>
<th>code of g()</th>
</tr>
</thead>
</table>

- 6 cache misses: g, f, h, f, h, f, h
- 2 cache misses: ▲ ▲
Change the variable declaration order

```c
struct S {
    int x1;
    int x2[200];
    int x3;
} obj[100];
for(...) {
    ... = obj[i].x1 + obj[i].x3;
}
```

```c
struct S {
    int x1;
    int x3;
    int x2[200];
} obj[100];
for(...) {
    ... = obj[i].x1 + obj[i].x3;
}
```

Improved spatial locality

- Now x1 and x3 likely reside in same cache line
- Access to x3 will always hit in the cache
Data Layout Optimization

- Change AOS (array of structs) to SOA (struct of arrays)

```
struct Point {
    int x;
    int y;
} points[100];
for(...) {
    ... = points[i].x * 2;
}
for(...) {
    ... = pointsj[i].y * 2;
}
```

```
struct Point {
    int x[100];
    int y[100];
} points;
for(...) {
    ... = points.x[i] * 2;
}
for(...) {
    ... = points.y[i] * 2;
}
```

- Improved spatial locality for accesses to ‘x’s and ‘y’s
- More efficient vectorization (no need to gather/scatter)
  - Gather: Load data from dispersed locations into vector unit
  - Scatter: Store data from dispersed locations into vector unit
Code-Related Optimizations
Code-Related Optimizations

- Modifying code  
  e.g. strength reduction
  \[ A = 2a; \quad \equiv \quad A = a \ll 1; \]

- Deleting code  
  e.g. dead code elimination
  \[ A = 2; A = y; \quad \equiv \quad A = y; \]

- Moving code  
  e.g. code scheduling
  \[ A = x \times y; B = A + 1; C = y; \quad \equiv \quad A = x \times y; C = y; B = A + 1; \]
  (Now \( C = y \); can execute while waiting for \( A = x \times y; \))

- Inserting code  
  e.g. data prefetching
  \[
  \text{while (p!=NULL)} \\
  \{ \text{process(p); p=p->next; } \} \\
  \equiv \\
  \text{while (p!=NULL)} \\
  \{ \text{prefetch(p->next); process(p); p=p->next; } \}
  \]
  (Now access to \( p->next \) is likely to hit in cache)
Optimization Categories

- Optimize at what representation level?
  - Source level — represented using AST
  - IR level — represented using low-level IR (3-address code)
  - Machine level — represented using machine-code IR

- Optimize across control flow?
  - Local optimization — scope within straight line code
    - Cannot be interrupted by any incoming or outgoing jumps
    - All instructions in scope executed exactly once — simple
  - Global optimization — scope across control structures
    - Scope can contain if / while / for statements
    - Some insts may not execute, or even execute multiple times

- Optimize across procedures?
  - Intra-procedural — scope within individual procedure
  - Inter-procedural — scope across different procedures
    - Analyzes other procedures called within scope to do better
Local Optimizations
Local Optimizations

- Optimizations where the scope includes no control flow
  - Limited in scope but can still do useful things

- **Strength Reduction**
  - The idea is to replace expensive operations (multiplication, division) by less expensive operations (add, sub, shift, mov)
  - Some are redundant and thus can be deleted
    - e.g. $x=x+0; y=y*1$
  - Some can be simplified
    - e.g. $x=x*0; y=y*8$
    - can be replaced by $x=0; y=y\ll3$
  - Typically performed at machine code level since knowledge of machine is required (e.g. multiplication is expensive)
More Local Optimizations

**Constant folding**
- Operations on constants can be computed at compile time
- In general, if \( x = y \text{ op } z \) and \( y \) and \( z \) are constants then compute at compile time and replace

**Example:**
```c
#define LEN 100
x = 2 * LEN;
if (LEN < 0) print("error");
```

Can be transformed to ...

```c
x = 200;
if (false) print("error");
```

- Performed at IR level since beneficial regardless of machine
Global Optimizations and Control Flow Analysis
Global optimization can work across control flow

- In effect, scope of optimization is entire function
- (Not global as in entire program as the name implies)
- E.g. Constant Propagation:
  - Replace variables with constants if value is known
    - X = 7;
    - ... // Contains jumps
    - Y = X + 3; // Can be replaced by Y = 10;
  - Local Constant Propagation works only on straightline code
  - Global Constant Propagation works even with jumps
    but needs to know the control flow between statements
    E.g. whether X = 7; is guaranteed to happen before Y = X+3;

Global optimization requires control flow analysis

- **Control flow analysis**: Compiler analysis that determines
  flow of execution between statements in function
- Constructs a control flow graph that describes the flow
A **basic block** is a maximal sequence of instructions that
- Except the first instruction, there are no other labels;
- Except the last instruction, there are no jumps;

Therefore,
- Can only jump into the beginning of a block
- Can only jump out at the end of a block

Are units of control flow that cannot be divided further
- All instructions in basic block execute or none at all
A control flow graph is a directed graph in which
- Nodes are basic blocks
- Edges represent flow of execution
  - Control statements such as if-then-else, while-loop, for-loop introduce control flow edges

CFG is widely used to represent a program

CFG is widely used for program analysis, especially for global analysis/optimization
Example

L1; t := 2 * x;
    w := t + y;
    if (w < 0) goto L3
L2: ...
...
L3: w := -w
...

L2: ...
...

L3: w := -w
...

L1; t := 2 * x;
    w := t + y;
    if (w < 0) goto L3
Construction of CFG

- **Step 1:** partition code into basic blocks
  - Identify **leader** instructions that are
    - the first instruction of a program, or
    - target instructions of jump instructions, or
    - instructions immediately following jump instructions
  - A basic block consists of a leader instruction and subsequent instructions before the next leader

- **Step 2:** add an edge between basic blocks B1 and B2 if
  - B2 follows B1, and B1 may "fall through" to B2
    - B1 ends with a conditional jump to another basic block
    - B1 ends with a non-jump instruction (B2 is a target of a jump)
    - Note: if B1 ends in an unconditional jump, cannot fall through
  - B2 doesn’t follow B1, but B1 ends with a jump to B2
Example

01. A=4
02. T1=A*B
03. L1: T2=T1/C
04: if (T2<W) goto L2
05: M=T1*K
06: T3=M+1
07: L2: H=I
08: M=T3-H
09: if (T3>0) goto L3
10: goto L1
11: L3: halt
Example

1. A=4
2. T1=A*B
3. L1: T2=T1/C
4. if (T2<W) goto L2
5. M=T1*K
6. T3=M+1
7. L2: H=I
8. M=T3-H
9. if (T3>0) goto L3
10. goto L1
11. L3: halt
Global Optimizations

- Extend optimizations to flow of control, i.e. CFG

- How do we know it is OK to globally propagate constants?
Correctness

- Optimization must be stopped if incorrect in even one path

- To replace \( x \) by a constant \( C \) **correctly**, we must know:
  - Along **all paths**, the last assignment to \( X \) is “\( X := C \)”
  - **All paths** often include branches and even loops
    - Potentially, there can be an infinite number of paths
    - Hard for compiler to always know which paths are possible
      (E.g. It may be \( B > 0 \) is always true at runtime)
Global Optimizations Need to be Conservative

Many compiler optimizations depend on knowing some property X at a particular point in program execution.

- Need to prove at that point property X holds along all paths.
Global Optimizations Need to be Conservative

Many compiler optimizations depend on knowing some property $X$ at a particular point in program execution:

- Need to prove at that point property $X$ holds along all paths
- Need to be **conservative** to ensure correctness
  - An optimization is enabled only when $X$ is definitely true
  - If not sure if it is true or not, it is safe to say don’t know
  - If analysis result is don’t know, no optimization done
  - May lose opt. opportunities but guarantees correctness

E.g. Global Constant Propagation (GCP):

```plaintext
X = 7;
...
Y = X + 3; // Replace by Y = 10, if X didn’t change
```

Needs knowledge of data flow, as well as control flow (Whether data flow is interrupted between points A and B)
Global Optimizations Need to be Conservative

Many compiler optimizations depend on knowing some property X at a particular point in program execution

- Need to prove at that point property X holds along all paths
- Need to be conservative to ensure correctness
  - An optimization is enabled only when X is definitely true
  - If not sure if it is true or not, it is safe to say don’t know
  - If analysis result is don’t know, no optimization done
  - May lose opt. opportunities but guarantees correctness

Property X often involves data flow of program

- E.g. Global Constant Propagation (GCP):
  - X = 7;
  - ...
  - Y = X + 3; // Replace by Y = 10, if X didn’t change
  - Needs knowledge of data flow, as well as control flow (Whether data flow is interrupted between points A and B)
Global Optimizations and Data Flow Analysis
Most optimizations rely on a property at given point

- For Global Constant Propagation (GCP):
  \[ A = B + C; \quad \text{// Property: \{A=?, B=10, C=?\}} \]
  After optimization:
  \[ A = 10 + C; \quad \text{// Property: \{A=?, B=10, C=?\}} \]

For this discussion, let’s call these properties \textit{values}

\textbf{Dataflow analysis}: Compiler analysis that calculates values for each point in a program

- Values get propagated from one statement to the next
- Statements can modify values (for GCP, assigning to vars)
- Requires CFG since values flow through control flow edges

\textbf{Dataflow analysis framework}: A framework for dataflow analysis that guarantees correctness for all paths

- Does \textit{not} traverse all possible paths (could be infinite)
- To be feasible, makes \textit{conservative} approximations
Overview of algorithm

- Initialize each point with the most optimistic values
  - For GCP: most optimistic ⇒ all vars are uninitialized
    (That means compiler can replace var with any constant)
- Propagate each value one step through one statement
  - For GCP: \{A=?, B=?\} through A=1; results in \{A=1, B=?\}
  - Each step refines values to be progressively pessimistic
- Iteratively propagate until a fixed point is reached

Two questions, which will be answered later:

- Does a fixed point exist (is it guaranteed to stop)?
- Does fixed point give a correct and precise set of values?

Rather than bore you with math, let’s first learn by example: global constant propagation (GCP)
Global Constant Propagation (GCP)

- Let’s apply framework to compute values for GCP
- Let’s use following notation to express the state of a var:
  - $x=*$  // not initialized (most optimistic)
  - $x=1, x=2, \ldots$  // a constant value (in between)
  - $x=#$  // not provably constant (most pessimistic)
- All values start as $x=*$ and are iteratively refined
  - Until they stabilize and reach a fixed point
In this example, constants can be propagated to $X+1$, $2*X$.

Statements visited in reverse postorder (predecessor first):

- $X:=3$;
- if ($B>0$)
- $Y:=Z+W$;
- $X:=4$;
- $Y:=0$;
- $X:=X+1$;
- $A:=2*X$;
In this example, constants can be propagated to $X+1$, $2*X$.

Statements visited in reverse postorder (predecessor first):

1. $A:=2*X$;
2. $X:=X+1$;
3. $Y:=0$;
4. $X:=4$;
5. $Y:=Z+W$;
6. $X:=3$;
7. \( \text{if } (B>0) \)

Diagram:

- Node 1: $X:=3$;
- Node 2: \( \text{if } (B>0) \)
- Node 3: $Y:=Z+W$;
- Node 4: $X:=4$;
- Node 5: $Y:=0$;
- Node 6: $X:=X+1$;
- Node 7: $A:=2*X$;
In this example, constants can be propagated to \( X+1 \), \( 2X \).

Statements visited in reverse postorder (predecessor first):

1. \( X \) = 3
2. \( Y = Z + W \)
3. \( A = 2X \)
4. \( X = 4 \)
5. \( Y = 0 \)
6. \( X = X + 1 \)
In this example, constants can be propagated to $X+1$, $2*X$.

Statements visited in reverse postorder (predecessor first):

- $X:=3$;
- if $(B>0)$
- $Y:=Z+W$;
- $X:=4$;
- $Y:=0$;
- $X:=X+1$;
- $A:=2*X$;
In this example, constants can be propagated to $X+1$, $2*X$.

Statements visited in reverse postorder (predecessor first):

- $X:=A:=2*X$;
- $X:=X+1$;
- $X:=4$;
- $Y:=Z+W$;
- $X:=3$;
- $X:=3$;
- $X:=3$;
- $X:=3$;
- $X:=3$;
- $X:=*$;
- $X:=*$;
- $X:=*$;
In this example, constants can be propagated to $X+1$, $2X$.

Statements visited in reverse postorder (predecessor first):

- $X := 3$
- $if (B > 0)$
- $Y := Z + W$
- $X := 4$
- $X := X + 1$
- $Y := 0$
- $X := 2X$
- $X := *$
In this example, constants can be propagated to $X+1$, $2X$.

Statements visited in reverse postorder (predecessor first):

- $X := 3$
- If ($B > 0$)
- $Y := Z + W$
- $X := 4$
- $A := 2X$
- $X := 0$
- $X := X + 1$
- $X := 4$
- $X := 3$
- $X := 3$
- $X := *$

Fixed Point
In this example, constants can be propagated to $X+1$, $2X$.

Statements visited in reverse postorder (predecessor first):

1. $A := 2 \times 4$
2. $X := 3 + 1$
3. $Y := 0$
4. If $(B > 0)$
5. $X := 4$
6. $X := 3$
7. $X := 3$
8. $X := 3$
9. $Y := Z + W$
10. $X := 4$
11. $X := 3$
12. $X := 3$
13. $X := *$
14. $X := 3$
Example GCP with Loop (Iteration 1)

In this example, loop prevents any constant propagation.

Statements visited in reverse postorder (predecessor first):

- $X := 3$
- if ($B > 0$)
- $Y := Z + W$
- $X := 4$
- $Y := 0$
- $X := X + 1$
- $A := 2X$
In this example, loop prevents any constant propagation.

Statements visited in reverse postorder (predecessor first):

\[ X := 3; \]
\[ \text{if } (B > 0) \]
\[ Y := Z + W; \]
\[ X := 4; \]
\[ X := X + 1; \]
\[ Y := 0; \]
\[ X := X + 1; \]
\[ A := 2 \times X; \]
In this example, loop prevents any constant propagation.

Statements visited in reverse postorder (predecessor first):

- \( X := \ast \)
- \( X = 3 \)
- \( X := 3; \) If \((B > 0)\)
- \( Y := Z + W; \) \( X := 4; \)
- \( Y := 0; \) \( X := X + 1 \)
- \( A := 2 \ast X; \)
In this example, loop prevents any constant propagation.

Statements visited in reverse postorder (predecessor first):

- $X := 3$
- If $(B > 0)$
- $Y := Z + W$
- $X := 4$
- $Y := 0$
- $X := X + 1$
- $A := 2X$
In this example, loop prevents any constant propagation

Statements visited in reverse postorder (predecessor first)
In this example, loop prevents any constant propagation.

Statements visited in reverse postorder (predecessor first)

```
X:=3;
if (B>0)
Y:=Z+W;
X:=4;
Y:=0;
X:=X+1
A:=2*X;
```
In this example, loop prevents any constant propagation

Statements visited in reverse postorder (predecessor first)

```
X := 3;
if (B > 0) {
  Y := Z + W;
  X := 4;
  Y := 0;
  X := X + 1;
}
A := 2 * X;
```
In this example, loop prevents any constant propagation

Statements visited in reverse postorder (predecessor first)

\[ X := 3; \]
\[ \text{if } (B > 0) \]
\[ Y := Z + W; \]
\[ X := 4; \]
\[ X := 4; \]
\[ Y := 0; \]
\[ X := X + 1; \]

\[ A := 2 \times 4; \]

\[ X := 3; \]
\[ X := 3; \]
\[ X := 3; \]
\[ X := 3; \]
\[ X := 3; \]

\[ X := 4; \]
\[ X := 4; \]
\[ X := 4; \]
\[ X := 4; \]
\[ X := 4; \]

\[ X := \ast; \]
\[ X := \ast; \]
\[ X := \ast; \]
\[ X := \ast; \]
\[ X := \ast; \]

\[ X := *; \]
\[ X := *; \]
\[ X := *; \]
\[ X := *; \]
\[ X := *; \]

NAT SO FAST!
Example GCP with Loop (Iteration 2)

- Fixed point is not reached in iteration 1 due to backedge
- Must do another iteration to reach fixed point

```
X:=3;
if (B>0)
  Y:=Z+W;
  X:=4;
  Y:=0;
  X:=X+1
A:=2*X;
```

Diagram:

- X
- Y
- Z
- W
- X:=3
- Y:=Z+W
- X:=4
- Y:=0
- X:=X+1
- A:=2*X
Example GCP with Loop (Iteration 2)

Fixed point is not reached in iteration 1 due to backedge
Must do another iteration to reach fixed point

```plaintext
X:=3;
if (B>0) {
  Y:=Z+W;
  X:=4;
  Y:=0;
  X:=X+1
}
A:=2*X;
```
Example GCP with Loop (Iteration 2)

- Fixed point is not reached in iteration 1 due to backedge
- Must do another iteration to reach fixed point

```plaintext
X := 3;  
if (B > 0)  
Y := Z + W;  
X := 4;  
Y := 0;  
X := X + 1
```

**Initialization:**
- X = ∗
- X = 3
- A := 2 * X

**Iteration 1:**
- X = 4

**Iteration 2:**
- Y := 0;  
X := X + 1

**Final State:**
- X = 4
Fixed point is not reached in iteration 1 due to backedge
Must do another iteration to reach fixed point

Example GCP with Loop (Iteration 2)
Example GCP with Loop (Iteration 2)

- Fixed point is not reached in iteration 1 due to backedge
- Must do another iteration to reach fixed point

```
X := 3;
if (B > 0)
    Y := Z + W;
    X := 4;
    Y := 0;
    X := X + 1
A := 2 * X;
```
Components of a Dataflow Analysis Framework

Components: Defined by \( \{ D, V, \land, F : V \rightarrow V \} \)

- **D**: Direction of propagation (forwards or backwards)
- **V**: Set of values (depends on analyzed property)
- **\( \land \)**: Meet operator \( (V \land V \rightarrow V) \)
  - Defines behavior when values meet at control flow merges
- **F**: Transfer function \( F : V \rightarrow V \)
  - Defines behavior of each basic block (statement)

Once **D**, **V**, **\( \land \)**, and **F** are defined, the framework takes care of the rest.

Each type of dataflow analysis will define them differently.

How are they defined for GCP?
Components of a Dataflow Analysis Framework

- Components: Defined by \( \{D, V, \land, F: V \rightarrow V\} \)
  - \(D\): Direction of propagation (forwards or backwards)
  - \(V\): Set of values (depends on analyzed property)
  - \(\land\): Meet operator \((V \land V \rightarrow V)\)
    - Defines behavior when values meet at control flow merges
  - \(F\): Transfer function \(F: V \rightarrow V\)
    - Defines behavior of each basic block (statement)

- Once \(D, V, \land, F\) are defined, framework takes care of rest
  - Each type of dataflow analysis will define them differently
  - How are they defined for GCP?
What is V?

Definition: Set of values in property under analysis

Property for GCP:
- What are the variables with constant values?
- And what are their values at the given point?
What is V?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are there values at the given point?
- A given variable can be in one of the following states:
  - \( x=* \) // not initialized (most optimistic)
  - \( x=1, x=2, ... \) // a constant value (in between)
  - \( x=# \) // not provably constant (most pessimistic)
What is $V$?

- Definition: Set of values in property under analysis
- Property for GCP:
  - What are the variables with constant values?
  - And what are their values at the given point?
- A given variable can be in one of the following states:
  - $x=*$ // not initialized (most optimistic)
  - $x=1, x=2, \ldots$ // a constant value (in between)
  - $x=#$ // not provably constant (most pessimistic)
- $V$ for GCP: Set of values where each value is the set of variables and their respective states.
- Examples of values in $V$: $\{x=*, y=10, z=#\}$, $\{x=1, y=\#, z=5\}$
- Goal for GCP is to assign a value to each point in program
What is $\land$?

$\land$: Meet operator $(V \land V \rightarrow V)$
- Defines behavior when values meet at control flow merges
- Given
  - $V_{in}(B)$ — value at the entry of basic block $B$
  - $V_{out}(B)$ — value at the exit of basic block $B$
- $V_{in}(B) = \land V_{out}(P)$ for each $P$, where $P$ is a predecessor of $B$

Example of $\land$ operator for GCP:
\{x=\*, y=2, z=3\} $\land$ \{x=1, y=2, z=10\} = \{x=1, y=2, z=\#\}

Why is $z=\#$ after the meet?
Why is $x=1$ after the meet?

$\land$ operator applied to values in $V$ must form a Semilattice
- **Semilattice**: Partial ordering of values with a lower bound
- Meet-semilattice to be exact but let’s just call it semilattice
Semilattice for GCP $\land$ operator (on just one variable):

$$\{x=\ast\}$$

$$\{x=-1\}$$

$$\{x=0\}$$

$$\{x=1\}$$

$$\{x=\#\}$$

$\land$ operator is defined by **Greatest Lower Bound (GLB)**

- $\{x=\ast\} \land \{x=1\} = \{x=1\}$
- $\{x=0\} \land \{x=1\} = \{x=\#\}$

What makes this a semilattice?

1. There is a single lower bound ($\{x=\#\}$)
2. It’s a partial order (values monotonically head downwards)

Note: some operators are not meet operators (e.g. add)
In a semilattice, there are two special values: \( \top \) and \( \bot \).
In a semilattice, there are two special values: \( \top \) and \( \bot \)

- \( \top \): Called **Top Value** (at top of semilattice)
  - Initial value when analysis begins (most optimistic)
  - For GCP: \{x=*, y=*, z=*\} (all vars uninitialized)
  - Value is refined in the course of analysis

- \( \bot \): Called **Bottom Value** (at bottom of semilattice)
  - Value which can be refined no further (most pessimistic)
  - For GCP: \{x=#, y=#, z=#\} (no vars provably constant)
In a semilattice, there are two special values: $\top$ and $\bot$.

$\top$: Called **Top Value** (at top of semilattice)
- Initial value when analysis begins (most optimistic)
- For GCP: \{x=*, y=*, z=*\} (all vars uninitializated)
- Value is refined in the course of analysis

$\bot$: Called **Bottom Value** (at bottom of semilattice)
- Value which can be refined no further (most pessimistic)
- For GCP: \{x=\#, y=\#, z=\#\} (no vars provably constant)

Properties of semilattice guarantee values stabilize
- Partial order guarantees value changes always downwards
- Lower bound ($\bot$) guarantees there is a termination point
What is F?

- **F**: Transfer function \((F: V \rightarrow V)\)
  - Defines what happens to value within a basic block
  - Given
    - \(V_{\text{in}}(B)\) — value at the entry of basic block \(B\)
    - \(V_{\text{out}}(B)\) — value at the exit of basic block \(B\)
  - \(V_{\text{out}}(B) = F(V_{\text{in}}(B))\)

- **F** for GCP:
  \[
  V_{\text{out}}(B) = (V_{\text{in}}(B) - \text{DEF}_v(B)) \cup \text{DEF}_c(B)
  \]
  where \(\text{DEF}_v(B)\) = set of vars assigned with variables in \(B\)
  \(\text{DEF}_c(B)\) = set of vars assigned with constants in \(B\)

- Easier if you treat each statement as a basic block
  - No multiple defs and overlaps between \(\text{DEF}_v(B)\), \(\text{DEF}_c(B)\)
There are two modes of propagation: \textbf{F} and \textbf{∧}

\begin{itemize}
  \item \textbf{F} — propagates values through basic blocks
    \begin{itemize}
      \item Variables in DEF\textsubscript{v} are set to \#
      \item Variables in DEF\textsubscript{c} are set to constant value
    \end{itemize}
  \item \textbf{∧} operator — propagates values through CFG edges
    \begin{itemize}
      \item Merges values from multiple predecessor blocks
    \end{itemize}
\end{itemize}
What is D?

\[ D: \text{Direction of propagation (forwards or backwards)} \]

Forward Analysis

Backward Analysis
What is D?

- Values are propagated forward: **Forward Analysis**
- Values are propagated backward: **Backward Analysis**
- GCP is an example of a Forward Analysis
  - Starting from a constant definition, the ‘constantness’ of a variable propagates forward through CFG
- We will see an example of Backward Analysis soon
Forward Analysis Algorithm

- Pseudocode for Dataflow Analysis Framework (Forward)
  for (each basic block B) $V_{out}(B) = T$;
  $W = \{\text{all basic blocks}\}$;
  while ($W \neq \emptyset$) {
    $B = \text{choose basic block from } W$;
    $V_{in}(B) = \bigwedge_{P \text{ is a predecessor of } B} V_{out}(P)$
    $V_{out}(B) = F(V_{in}(B))$
    if ($V_{out}(B)$ is changed) $W = W \cup \{B\text{'s successors}\}$
  }

- $V, \bigwedge$ and $F$ defined differently for each type of analysis

- Will it eventually stop at a fixed point?
  - If there are loops, we may go through the loop many times
  - Is there a possibility of values changing forever?

- Will the fixed point give a correct and precise solution?
Termination Problem

- Existence of $\bot$ value ensures termination
  - Values start from $\top$ and can only go down in semilattice
  - Number of value changes is limited by height of semilattice

- Computational complexity ($V = \text{vars}$, $N = \text{basic blocks}$)
  - Each basic block can only change value $2 \times V$ times
    (Twice for each variable according to semilattice)
  - Maximal complexity: $O(2 \times V \times N) = O(V \times N)$
    - Each basic block can appear $2 \times V$ times in $W$ (work list)
  - Practical complexity: $O(N)$
    - With reverse postorder traversal of basic blocks,
      1. Blocks in straight-line code change only once
      2. Blocks in singly-nested loop change at most twice
         (Loop always converges on second traversal, as we saw)
      3. Blocks in $L$-nested loop change at most $L+1$ times
      4. $O(L \times N) = O(N)$, since typically $L \leq 3$
A few different types of solutions:

- IDEAL: Meet of all possible paths \( F_P \) to this point
  
  \[
  \text{IDEAL}(B) = \bigwedge P \text{ is possible path from ENTRY to } B \ \text{\( F_P(V_{\text{ENTRY}}) \)}
  \]

- MOP (Meet-Over-Paths): Meet of all paths in CFG
  
  \[
  \text{MOP}(B) = \bigwedge P \text{ is path in CFG from ENTRY to } B \ \text{\( F_P(V_{\text{ENTRY}}) \)}
  \]

- MFP (Maximum Fixed Point): Given iterative solution

\[\text{MFP} \leq \text{MOP} \leq \text{IDEAL} \text{ (in semilattice)}\]

- MOP \leq IDEAL: Why?
  - Paths in CFG is a superset of all possible paths
  - \( MOP = IDEAL \wedge V_{\text{never taken paths}} \leq IDEAL \) (since GLB)

- MFP \leq MOP: Why?
  - MFP stops only when fixed point is reached:
    - Covers all paths in MOP, even for limitless iterations
  - For GCP: sometimes MFP < MOP (next slide)

MFP is correct but not precise (in short conservative)
When is MFP $< \text{MOP}$?

- Assume $V_{ENTRY} \equiv \{ A = \ast, B = \ast, C = \ast \}$:

  - **P1**: $A=1$; $B=2$;
  - **P2**: $A=2$; $B=1$;
  - **B**: $C=A+B$;

- $\text{MOP} \equiv F_B(F_{P1}(V_{ENTRY})) \land F_B(F_{P2}(V_{ENTRY}))$
  
  $\equiv \{ A = 1, B = 2, C = 3 \} \land \{ A = 2, B = 1, C = 3 \}$
  
  $\equiv \{ A = \#, B = \#, C = 3 \}$

- $\text{MFP} \equiv F_B(F_{P1}(V_{ENTRY}) \land F_{P2}(V_{ENTRY}))$
  
  $\equiv F_B(\{ A = \#, B = \#, C = \ast \}) \equiv \{ A = \#, B = \#, C = \# \}$

- $\text{F}$ for GCP is not **distributive** (Refer to Chapter 9.3)
Once constants have been globally propagated, we would like to eliminate the dead code:

\[
\begin{align*}
x &:= 3; \\
\text{if } (b > 0) &\quad \text{if } (b > 0) \\
y &:= z + w; \\
z &:= 2x;
\end{align*}
\]
Once constants have been globally propagated, we would like to eliminate the dead code:

\[ x := 3; \]
\[ \text{if (b>0)} \]
\[ y := z + w; \]
\[ y := 0; \]
\[ z := 2 \times 3; \]
Once constants have been globally propagated, we would like to eliminate the dead code

```
x := 3;
if (b > 0)
    y := z + w;
y := 0;
z := 2*3;
```
A **dead statement** calculates a value that is not used later.

Otherwise, it is a **live statement**.

In the example, the 1st statement is dead, the 2nd statement is live.
Global Liveness Analysis (GLA)

- A variable $X$ is live at statement $S$ if
  - There exists a statement $S_2$ after $S$ that uses $X$
  - There is a path from $S$ to $S_2$
  - There is no intervening assignment to $X$ between $S$ and $S_2$
Global Liveness Analysis (GLA)

- A variable X is live at statement S if
  - There exists a statement S2 after S that uses X
  - There is a path from S to S2
  - There is no intervening assignment to X between S and S2

Again a dataflow analysis framework can be applied

- What is \( D, V, \land, F : V \rightarrow V \) in this context?

What is \( D \)?

- Liveness Analysis is a Backward Analysis
  - Starting from a use, the ‘liveness’ of a variable propagates backward through CFG
  - Changes direction of \( \land \) operator and transfer function
Forward and Backward Analysis Again

Forward Analysis

Backward Analysis
What is $V$?

- **Definition**: Set of values in property under analysis
  - $V$ for GLA: Each value is a set of live variables
  - Example values: $\{x, y, z\}$, $\{y\}$

- $\top$: initial value at the beginning
  - $\top$ for GLA = $\{\}$
  - Start with assumption that no variables are live

- $\bot$: the don’t know value
  - $\bot$ for GLA = *all variables in function*
  - Meaning: none of the variables are provably dead
What is $\land$?

$\land$: Meet operator ($V \land V \rightarrow V$) for backward analysis

- Defines behavior when values meet at control divergence
- Given
  - $V_{in}(B)$ — value at the entry of basic block $B$
  - $V_{out}(B)$ — value at the exit of basic block $B$
- $V_{out}(B) = \land V_{in}(S)$ for each $S$, where $S$ is successor of $B$
- Note the reversal in direction! GLA is a backward analysis.

$\land$ operator for GLA:

- Meet operator is a simple union $\cup$
- Example: $\{x, y\} \land \{y, z\} = \{x, y\} \cup \{y, z\} = \{x, y, z\}$
- Union operator monotonically increases set (a partial order), hence values form a semilattice from $\top$ to $\bot$
What is F?

F: Transfer function (F: V → V) for backward analysis

- Defines what happens to value within a basic block
- Given
  - \( V_{in}(B) \) — value at the entry of basic block \( B \)
  - \( V_{out}(B) \) — value at the exit of basic block \( B \)
- \( V_{in}(B) = F( V_{out}(B) ) \)
- Again note the reversal in direction!

F for GLA:

\[
V_{in}(B) = ( V_{out}(B) - DEF(B) ) \cup USE(B)
\]

where DEF(B) contains variable definitions in B

USE(B) contains variable uses in B

Easier to reason about if you treat each individual statement as a basic block
Liveness Example

b = b + c

a = d + 1;
Liveness Example

\[ b\leftarrow b+c \]
\[ a\leftarrow d+1; \]
Liveness Example

\[ b = b + c \]
\[ a = d + 1; \]

\[ V_{\text{in}}(B1) \]
\[ V_{\text{out}}(B1) \]
\[ V_{\text{in}}(B2) \]

\[ V_{\text{out}}(B2) \]

\[ V_{\text{in}}(B3) \]

\[ V_{\text{out}}(B3) = \{ a, b \} \]
Liveness Example

Two sets:
- DEF = \{a\}
- USE = \{d\}

Diagram:
- \( b = b + c \)
- \( a = d + 1; \)
- \( V_{in}(B2) \)
- \( V_{out}(B2) \)
- \( V_{out}(B3) = \{a, b\} \)
- \( V_{out}(B3) \)
Liveness Example

(b + c)

\[ b = b + c \]

\[ a = d + 1; \]

Two sets:
- \( \text{DEF} = \{a\} \)
- \( \text{USE} = \{d\} \)
Liveness Example

\[ b = b + c \]
\[ a = d + 1; \]

Two sets:
- **DEF** = \{a\}
- **USE** = \{d\}

\[ V_{in}(B1) \]
\[ V_{in}(B2) = \{b, c\} \]
\[ V_{in}(B3) = \{b, d\} \]

\[ V_{out}(B2) \]
\[ V_{out}(B3) = \{a, b\} \]
Liveness Example

\[ b = b + c \]
\[ a = d + 1; \]

Two sets:
- **DEF** = \{a\}
- **USE** = \{d\}

Graph:
- \( V_{in}(B1) \)
- \( V_{out}(B1) = \{b, c, d\} \)
- \( V_{in}(B2) = \{b, c\} \)
- \( V_{out}(B2) \)
- \( V_{in}(B3) = \{b, d\} \)
- \( V_{out}(B3) = \{a, b\} \)
Backward Analysis Algorithm

Pseudocode for Dataflow Analysis Framework (Backward)
for (each basic block B) \( V_{in}(B) = \top \);
\( W = \{ \text{all basic blocks} \} \);
while (\( W \neq \emptyset \)) {
    B = \text{choose basic block from } W;
    \( V_{out}(B) = \bigwedge S \text{ is a successor of } B \ V_{in}(S) \)
    \( V_{in}(B) = F(V_{out}(B)) \)
    if (\( V_{in}(B) \) is changed) \( W = W \cup \{ B's \text{ predecessors} \} \)
}

Note the reversal in direction compared to forward analysis

Will backward analysis for GLA eventually stop?
- Again properties of semilattice ensures termination
- Value can change \( V \) times, where \( V \) is number of vars
- Maximal complexity: \( O(V \times N) \)
- Practical complexity: \( O(N) \), with postorder traversal
Is GLA Precise?

- For GLA, MFP = MOP \leq IDEAL
- MOP \leq IDEAL: CFG is a superset of all paths (like GCP)
- MFP = MOP: Why?
  - MFP emulates all paths in MOP (like GCP)
  - Unlike GCP, transfer function $F$ for GLA is **distributive**
    - $MOP \equiv F_B(F_{P_1}(V) \land F_B(F_{P_2}(V))$
    - $\equiv F_B(F_{P_1}(V) \land F_{P_2}(V)) \equiv MFP$
- If MOP = IDEAL, GLA is precise
Comparison of GCP and GLA

- **D**: Direction of propagation
  - GCP: Forward
  - GLA: Backward

- **V**: Set of values propagated
  - GCP: Set of variables with constant values
  - GLA: Set of live variables

- **∧**: Meet operator
  - GCP: Given by semilattice (Top $\rightarrow$ Constant $\rightarrow$ Bottom)
  - GLA: Simply the set union operator

- **F**: Transfer function
  - GCP: - var defs to variables, + vars defs to constants
  - GLA: - var defs, + var uses
Application of Liveness Analysis

- Global dead code elimination is based on GLA
  - Dead code detection
    - `x = ...;` is dead code if `x` is dead after this statement
    - Dead statement can be deleted from the program

- Global register allocation is also based on GLA
  - Only live variables are placed in registers
  - Registers holding dead variables can be reused
Register Allocation
What is Register Allocation?

- Process of assigning (a large number of) variables to (a small number of) CPU registers

- Registers are fast
  - access to memory: 100s of cycles
  - access to cache: a few to 10s of cycles
  - access to registers: 1 cycle

- But registers are limited in number
  - x86: 8 regs, MIPS: 32 regs, ARM: 32 regs ... 

- Goals of register allocation:
  - Keep frequently accessed variables in registers
  - Keep variables in registers only as long as they are live
Allocate registers basic block by basic block

- Makes decisions on a per-block basis (hence ‘local’)
- Uses results of Global Liveness Analysis for decisions

Requires only a single scan through each basic block

- Keeps track of two tables:
  - Register Table: which regs are in use and which available
  - Address Table: location(s) where each variable is stored
    (locations can be: register, stack memory, static memory)

- Initially, no regs in use and all vars stored in memory
  - Local variables, temporary variables ⇒ stack memory
  - Global variables ⇒ static memory

- During scan, do below for every use of variable:
  1. If var already in reg according to Address Table, no action
  2. If not, find available reg from Register Table and allocate
  3. If no available regs, select reg for displacement

Q: Which register should be displaced?
Local Register Allocation - Displacement

- Choose regs that can be recycled without saving value:
  - Register whose value is no longer live (given by GLA)
  - Register whose value has a copy in another location

- If none exist, choose reg to be spilled to memory

- **Spill**: storing variable in original memory location
  - Original memory location: stack or static memory
  - Store instruction to memory generated at point of spill
  - All uses of variable onwards must load from memory

- At end of scan, spill all live registers at end of block
  - To make all regs available for next basic block allocation
  - Allows next block allocation to remain 'local'
  - Causes unnecessary spills at basic block boundaries

- So what if we don't spill? Not as simple as you think
  - On control flow merges, what if two blocks have differing allocations of vars to regs?
  - ⇒ Must somehow reconcile
  - Global allocation decisions required to minimize the above
Local Register Allocation - Displacement

- Choose regs that can be recycled without saving value:
  - Register whose value is no longer live (given by GLA)
  - Register whose value has a copy in another location

- If none exist, choose reg to be spilled to memory

- **Spill**: storing variable in original memory location
  - Original memory location: stack or static memory
  - Store instruction to memory generated at point of spill
  - All uses of variable onwards must load from memory

- At end of scan, spill all live registers at end of block
  - To make all regs available for next basic block allocation
    - Allows next block allocation to remain ‘local’
    - Causes unnecessary spills at basic block boundaries
  - So what if we don’t spill? Not as simple as you think
    - On control flow merges, what if two blocks have differing allocations of vars to regs? Must somehow reconcile
    - Global allocation decisions required to minimize the above
Global Register Allocation

- Makes global decisions about register allocation such that
  - Var to reg mappings remain consistent across blocks
  - Structure of CFG is taken into account on decisions

- Relies on Global Liveness Analysis just like local allocation

- Three well-known register allocation algorithms
  1. Graph coloring allocator
  2. Linear scan allocator
  3. ILP (Integer Linear Programming) allocator
Algorithm steps:

1. Identify live range interference using GLA
2. Build register interference graph (RIG)
   - Node represents a variable
   - Edge represents overlap in live ranges between two vars
3. Attempt K-coloring of the graph
   - K is the number of available registers
   - Color each node such that adjacent nodes are different
4. On failure, spill a variable and go back to 3.
   - Spilling var to memory removes it from the graph
   - Which var decided using some heuristic. Considerations:
     Which var when spilt will simplify graph the most?
     Which var is the cheapest to spill (min. access frequency)?
Live Range Interference

**Live Range**: Set of program points where a variable is live

- Two live ranges interfere if there is an overlap
- Vars with interfering ranges cannot reside in same register
**Live Range Interference**

- **Live Range**: Set of program points where a variable is live
  - Two live ranges interfere if there is an overlap
  - Vars with interfering ranges cannot reside in the same register

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```

```
x := ...
y := ...
y := ...
y := ...
```
Live Range Interference

**Live Range**: Set of program points where a variable is live

- Two live ranges interfere if there is an overlap
- Vars with interfering ranges cannot reside in same register

We annotate each program point (between two statements) to explicitly show the interference.
Example of GLA and interfering live ranges

```plaintext
a := b + c;
d := -a;
e := d + f;
f := 2 * e;
b := b + e;
e := e - 1;
b := f + c;
```
Construct **Register Interference Graph (RIG)** such that

- Each node represents a variable
- An edge between two nodes $V_1$ and $V_2$ represents an interference in live ranges

Based on RIG,

- Two variables can be allocated in the same register if there is no edge between them
- Otherwise, they cannot be allocated in the same register
In the RIG for our example:

- b, c cannot be in the same register
- a, b, d can be in the same register
Allocating Registers using Graph Coloring

- Graph coloring is a theoretical problem where ...
  - A coloring of a graph is an assignment of colors to nodes such that nodes connected by an edge have different colors
  - A graph is k-colorable if it has a coloring with k colors

- Problem of register allocation maps to graph coloring
  - Once solved, k colors can be mapped back to k registers
  - If the graph is k-colorable, it’s k-register-allocatable
This is an coloring of our example RIG using 4 colors

There is no solution with less than 4 colors
Using the coloring result, map it back to the code

\[
\begin{align*}
\text{a} &:= \text{b} + \text{c}; \\
\text{d} &:= -\text{a}; \\
\text{e} &:= \text{d} + \text{f}; \\
\text{f} &:= 2 \times \text{e}; \\
\text{b} &:= \text{d} + \text{e}; \\
\text{e} &:= \text{e} - 1; \\
\text{b} &:= \text{f} + \text{c};
\end{align*}
\]
Using the coloring result, map it back to the code

\[
\begin{align*}
\text{a} & \rightarrow R1 \\
\text{b} & \rightarrow R2 \\
\text{c} & \rightarrow R3 \\
\text{d} & \rightarrow R2 \\
\text{e} & \rightarrow R1 \\
\text{f} & \rightarrow R4
\end{align*}
\]

\[
\begin{align*}
f & := 2 \times e; \\
b & := d + e; \\
e & := e - 1; \\
r1 & := r2 + r3; \\
r2 & := -r1; \\
r1 & := r2 + r4;
\end{align*}
\]
After Register Allocation

Using the coloring result, map it back to the code

\[ a \rightarrow R1 \]
\[ b \rightarrow R2 \]
\[ c \rightarrow R3 \]
\[ d \rightarrow R2 \]
\[ e \rightarrow R1 \]
\[ f \rightarrow R4 \]

\[ R1 := R2 + R3; \]
\[ R2 := -R1; \]
\[ R1 := R2 + R4; \]

\[ R4 := 2 \times R1; \]

\[ b := d + e; \]
\[ e := e - 1; \]

\[ b := f + c; \]
Using the coloring result, map it back to the code

a–R1
b–R2
c–R3
d–R2
e–R1
f–R4

R1 := R2 + R3;
R2 := -R1;
R1 := R2 + R4;

R4 := 2 * R1;

b := f + c;

R2 := R2 + R1;
R1 := R1 + 1;

R2 := R2 + R1;
R1 := R1 - 1;
Using the coloring result, map it back to the code

\[
\begin{align*}
R1 & := R2 + R3; \\
R2 & := -R1; \\
R1 & := R2 + R4; \\
R4 & := 2 \times R1; \\
R2 & := R4 + R3; \\
R2 & := R2 + R1; \\
R1 & := R1 - 1; \\
R2 & := R2 + R1; \\
R1 & := R1 - 1;
\end{align*}
\]
Determining whether a graph is k-colorable is NP-complete

- Therefore, problem of k-register allocation is NP-complete
- In practice: use heuristic polynomial algorithm that gives close to optimal allocations most of the time
- Chaitin’s graph coloring is a popular heuristic algorithm
  - E.g. most backends of GCC use Chaitin’s algorithm

What if k-register allocation does not exist?
- Spill a variable to memory to reduce RIG and try again
**Observation**: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what...
**Observation:** for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what
Observation: for a k-coloring problem, a node with k-1 neighbors can always be colored, no matter what
Observation: for a $k$-coloring problem, a node with $k-1$ neighbors can always be colored, no matter what...
**Corollary**: Given graph \( G \) for a \( k \)-coloring problem
- Let \( G' \) be graph after removing a node with <\( k \) neighbors
- If \( G' \) can be \( k \)-colored then \( G \) can be \( k \)-colored. How?
  1. Color \( G' \) then add back removed node
  2. Remaining node is always colorable since <\( k \) neighbors

**Idea**: Recursively solve for the simpler \( G' \) instead of \( G \)

**Algorithm**
- Phase 1: Repeat until there are no nodes left
  - Pick a node \( V \) with fewer than \( k \) neighbors
  - Push \( V \) on a stack and remove it and its edges from \( G \)
- Phase 2: Assign colors to nodes on the stack in LIFO order
  - Pop a node \( V \) from the stack
  - Pick a color for \( V \) that is different from its neighbors
    (Such a color is guaranteed to exist due to corollary)
Chaitin’s algorithm applied to our example where \( k=4 \)

\[
\begin{array}{c}
\text{Stack=\{\}} \\
\end{array}
\]
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a\}$
Chaitin’s algorithm applied to our example where $k=4$

Stack={$a$}
Chaitin’s algorithm applied to our example where $k=4$

Stack={$a,d$}

Graph:
- Nodes: $a, b, c, d, e, f$
- Edges: $a-b, a-c, a-d, b-c, b-d, c-d, e-f, e-c, e-b, e-d, f-c, f-b, f-d$
Chaitin’s Graph Coloring Example

Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d\}$
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b\}$
Chaitin’s algorithm applied to our example where k=4

Stack={a,d,b}
Chaitin’s Graph Coloring Example

Chaitin’s algorithm applied to our example where \( k=4 \)

Stack = \{a,d,b,c\}
Chaitin’s algorithm applied to our example where \( k=4 \)

Stack = \{a,d,b,c\}
Chaitin’s algorithm applied to our example where $k=4$

Stack={$a, d, b, c, e$}
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b,c,e\}$

$\bullet$
Chaitin’s algorithm applied to our example where $k=4$

Stack=$\{a,d,b,c,e,f\}$
Chaitin’s algorithm applied to our example where \( k=4 \)

\[
\text{Stack} = \{a,d,b,c,e,f\}
\]
Starting assigning colors to **f,e,c,b,d,a**

Stack={a,d,b,c,e,f}

Stack={a,d,b,c,e}

Stack={a,d,b,c}

Stack={a,d,b}

Stack={a,d}

Stack={a}

Stack={}
Is Chaitin’s Graph Coloring Optimal?

- According to Chaitin’s algorithm:
  Every node has 3 outgoing edges, thus it is not 3-colorable

![Graph Diagram]

![Graph Diagram]
Is Chaitin’s Graph Coloring Optimal?

- According to Chaitin’s algorithm:
  Every node has 3 outgoing edges, thus it is not 3-colorable

- However, it is 3-colorable as you can see above
- Chaitin’s algorithm is not optimal
What if Coloring Fails?

- Spill the variable to memory
  - Spilled var stays in memory and is not allocated a reg
  - e.g. To color the previous graph using 3 colors
    - Spill “f” into memory since it has the most edges
What if Coloring Fails?

- Spill the variable to memory
  - Spilled var stays in memory and is not allocated a reg
  - e.g. To color the previous graph using 3 colors
    - Spill “f” into memory since it has the most edges
What if Coloring Fails?

- Spill the variable to memory
  - Spilled var stays in memory and is not allocated a reg
  - e.g. To color the previous graph using 3 colors
    - Spill “f” into memory since it has the most edges
What if Coloring Fails?

- Spill the variable to memory
  - Spilled var stays in memory and is not allocated a reg
  - e.g. To color the previous graph using 3 colors
    - Spill “f” into memory since it has the most edges
What if Coloring Fails?

- Spill the variable to memory
  - Spilled var stays in memory and is not allocated a reg
  - e.g. To color the previous graph using 3 colors
    - Spill “f” into memory since it has the most edges
Linear Scan Register Allocation

- On-line compilers need to generate binary code quickly
  - Just-in-time compilation
  - Interactive environments e.g. IDE

- In these cases, it is beneficial to sacrifice code performance a bit for quicker compilation
  - A faster allocation algorithm
  - Not sacrificing too much in code quality

- Proposed in following publication:
  - Poletto, M., Sarkar, V., "Linear scan register allocation", in ACM Transactions on Programming Languages and Systems (TOPLAS), 1999
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each **live interval**

```
scan order

A
B
C
D
E
code
```
Linear Scan Register Allocation

- Layout the code in a certain linear order
- Do a single scan to allocate register for each **live interval**
Linear Scan Register Allocation

Layout the code in a certain linear order
Do a single scan to allocate register for each live interval

Allocate greedily at each numbered point in program
  A and D may use same register (same for B and E)
**Live Interval**: Smallest interval of code containing all live ranges in the given linear code layout

- Live range of \( a = \{B1, B3\} \), \( b = \{B2, B4\} \)
- If code layout is “B1,B3,B2,B4”, only 1 register is enough
  - Live interval of \( a = \{B1, B3\} \), \( b = \{B2, B4\} \)
- If code layout is “B1,B2,B3,B4”, then need 2 registers
  - Live interval of \( a = \{B1, B2, B3\} \), \( b = \{B2, B3, B4\} \)
Linear Scan Algorithm

Linear scan RA consists of four steps

S1. Order all instructions in linear fashion
   - Order affects quality of allocation but not correctness

S2. Calculate the set of live intervals
   - Each variable is given a live interval

S3. Greedily allocate register to each interval in order
   - If register not available, spill a live interval (variable)
   - When spilling, prefer longest remaining interval length

S4. Rewrite the code according to the allocation

Coloring vs Linear Scan Complexity Comparison

- Chaitin Coloring: \( O(V \times V) \), where \( V \) = number of vars
  - \( V \) steps to color graph with \( V \) nodes
  - At worst, \( V \) spills leading to \( V \) colorings

- Linear Scan: \( O(V) \), where \( V \) = number of vars
  - \( V \) allocations of live intervals in single scan
Register Allocation Time Comparison

- **Usage Counts**, **Linear Scan**, and **Graph Coloring** shown.
- Linear Scan allocation is always faster than Graph Coloring.
ILP-based Register Allocation

- Idea and steps:
  1. Convert RA problem to a ILP problem
  2. Solve ILP problem using widely known ILP solvers
  3. Map the ILP solution back to register assignment

- Goal: find “optimal” allocation
  - Chaitin graph coloring is a heuristic algorithm
  - Optimal (NP-complete) graph coloring algorithms exist, but still uses heuristics for spilling
  - ILP finds optimal allocation and placement of spill code

- Complexity restricts adoption for complex programs
  - Optimal ILP solution is NP-hard (similar to graph coloring)
  - Heuristic polynomial ILP solvers exist and are researched
What is Integer Linear Programming (ILP)?

- Integer Linear Programming (ILP)
  - Variables: a, b
  - Constraints:
    - $0 \leq a \leq 10$
    - $0 \leq b \leq 29$
    - $a + b \leq 36$
  - Goal function
    - minimize $f(a,b) = 3a + 4b$

- It is trivial if a and b can take real values
- It is NP hard if a and b can only take integer values
How to Convert Register Allocation to ILP?

- An example
  
  (9) ...
  (10) ... = b + a ;
  (11) ...

  ➢ Want to know to which register b should be allocated i.e. 
    load Rx, addr(b)

- Convert to an ILP problem

  ➢ assume there are four free registers R1, R2, R3, R4

S1: Define the variables in ILP

\[ V_{var(location)}^{Ri} \] — Whether var at location is allocated to Ri

\[ V^{R1}_{b(10)}, V^{R2}_{b(10)}, V^{R3}_{b(10)}, V^{R4}_{b(10)} \]

Value of 0 — not allocate to that register at the place
Value of 1 — is allocated to that register at the place
Converting Register Allocation to ILP

**S2:** Define constraints. E.g. for code (10) \( b + a, \)

- A register can hold at most one variable per place
  \[ V_{b(10)}^{R1} + V_{a(10)}^{R1} \leq 1, V_{b(10)}^{R2} + V_{a(10)}^{R2} \leq 1, \ldots \]
- A variable is allocated to exactly one register per place
  \[ V_{b(10)}^{R1} + V_{b(10)}^{R2} + V_{b(10)}^{R3} + V_{b(10)}^{R4} = 1 \]
  \[ V_{a(10)}^{R1} + V_{a(10)}^{R2} + V_{a(10)}^{R3} + V_{a(10)}^{R4} = 1 \]
- and many more ...

**S3:** Define goal function

- To minimize cost of memory operations for spilling:
  \[ f_{cost} = \sum V_{v(\text{loc})}^{stack} \ast U_{v(\text{loc})} \ast \text{exec\_count}(\text{loc}) \ast \text{LOAD}_{cost} \]
  \( V_{v(\text{loc})}^{stack} \): Whether \( v \) at \( \text{loc} \) is allocated to stack (spilled)
  \( U_{v(\text{loc})} \): Whether variable \( v \) is used at \( \text{loc} \)
  \( \text{exec\_count}(\text{loc}) \): Expected runtime execution count of \( \text{loc} \)
  \( \text{LOAD}_{cost} \): Cost of load instruction in given machine

**S4:** Run your favorite ILP solver
Conclusion

- Good Register Allocation is crucial to code quality
  - Accesses to memory are costly, even with caches

- Trade-offs between allocation time and code quality
  - Coloring: where quality is important
  - Linear scan: where allocation time is important (e.g. JIT)
  - ILP: where quality is paramount (e.g. real time systems)

- Previous compiler opts may affect allocation quality
  - E.g. Constant propagation may replace vars with constants
  - E.g. Dead code elimination may remove var uses altogether
  - Affect of compiler opts are intertwined and hard to separate
  - Finding optimal opt combinations is in itself research
  - Compilers package opts that typically go together into levels (e.g. -O1, -O2, -O3)
The END !