Optimization Overview

- Maximize the value of some *objective function*
- Execution time
  - Does the resulting program execute fast enough?
- Compilation time
  - Does the compiler produce code fast enough?
- Code size
  - Can the resulting code fit in a low-capacity embedded device?
- Power consumption
  - Will my code make datacenter owners happy?

**Remember** Correctness must never be implicated by optimizations (*lie*)
Pareto Optimization

- Optimization involves tradeoffs

![Diagram showing Pareto front with two options and satisfaction increases as one attribute increases.](image)
Optimizations to consider

▶ Things we have covered
  ▶ Constant folding/propagation
  ▶ Liveness analysis/dead code elimination
  ▶ Register allocation

▶ Things to consider for PA6
  ▶ Unboxing
  ▶ Copy propagation
  ▶ Function inlining
  ▶ Code motion/hoisting
  ▶ Loop invariants and unrolling
  ▶ Common subexpression elimination
  ▶ Static Single Assignment (SSA)
  ▶ Peephole optimizations

▶ Others to think about (but not for PA6)
  ▶ Whole-program and Link-time optimizations
  ▶ Optimizations performed by hardware
Dataflow Analysis: Reminder

- Dataflow analysis is a collection of techniques for compile-time reasoning about the run-time flow of values

- Local dataflow analysis: one basic block

- Global dataflow analysis: entire CFG of a whole method

- You have already seen specific examples of dataflow analysis: dead code elimination, register allocation, constant propagation
General Dataflow Analysis

Basic idea

▶ Setting up and solving systems of equations that relate information at various points in a program
  ▷ this is an iterative process

▶ Desired result is usually meet over all paths solution
  ▷ “What is true on every path from the entry?”
  ▷ “Can this happen on any path from the entry?”
Iterative Algorithms

- First, compute some local information within individual basic blocks
- Then, propagate local information along control flow edges
  - IN(B): some property on entry to basic block B
  - OUT(B): some property on exit from basic block B
  - Need to iterate until no changes

```plaintext
while change do
  change = false
  for each basic block
    apply equations updating IN or OUT
    if IN/OUT changes, set change to true
end
```
Recall: Liveness Analysis

<table>
<thead>
<tr>
<th>Block</th>
<th>LiveOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{a,b,e}</td>
</tr>
<tr>
<td>B2</td>
<td>{a,b,c,d,e}</td>
</tr>
<tr>
<td>B3</td>
<td>{a,b,c,d,e}</td>
</tr>
<tr>
<td>B4</td>
<td>{a,b,c,d,e}</td>
</tr>
<tr>
<td>B5</td>
<td>{a,b,d,e}</td>
</tr>
<tr>
<td>B6</td>
<td>{}</td>
</tr>
</tbody>
</table>

Compiler Construction
Types of Dataflow Analyses

 ► **Forward**
   - Determining information in Block $B$ requires propagating information from $PRED(B)$ in CFG
   - Examples: reachability, available expressions, constant propagation

 ► **Backward**
   - Determining information in Block $B$ requires propagating information from $SUCC(B)$ in CFG
   - Example:
Types of Dataflow Analyses

▶ **May**
  ▶ True on some path
    ▶ Example: live variable analysis — a variable is live if it may be used on some path
    ▶ Reflected as a set union in analysis: $LIVE_{out}$ for a block $B$ starts as the union of $LIVE_{in}$ for all successors of $B$

▶ **Must**
  ▶ True on all paths
    ▶ Example: constant propagation — a variable must have the same constant value on all incoming paths to propagate that value
    ▶ Reflected as a set intersection in analysis: all predecessors of block $B$ must have $\{x = y\}$ for some variable $x$ and some fixed constant $y$
## Dataflow Analyses

<table>
<thead>
<tr>
<th>Domain</th>
<th>AVAIL</th>
<th>REACHES</th>
<th>LIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Set of expressions</td>
<td>set of definitions</td>
<td>set of variables</td>
</tr>
<tr>
<td>Across</td>
<td>Forward</td>
<td>Forward</td>
<td>Backward</td>
</tr>
<tr>
<td>blocks</td>
<td>$\cap_{x \in \text{PRED}(B)} \text{OUT}(X)$</td>
<td>$\cup_{x \in \text{PRED}(B)} \text{OUT}(X)$</td>
<td>$\cup_{x \in \text{SUC}(B)} \text{IN}(X)$</td>
</tr>
<tr>
<td>Initialize</td>
<td>$\text{AVAIL}(B) = {E}$</td>
<td>$\text{REACH}(B) = {}$</td>
<td>$\text{LIVE}(B) = {}$</td>
</tr>
</tbody>
</table>

Dataflow Analysis can be generalized into a format like this

All that’s left is a way to compute information within each block (easy!)
Iterative Algorithms: Correctness

- First, compute some local information within individual basic blocks
- Then, propagate local information along control flow edges
  - IN(B): some property on entry to basic block B
  - OUT(B): some property on exit from basic block B
  - Need to iterate until no changes

\[
\begin{align*}
\text{while } \text{change} & \text{ do} \\
\text{change} & = \text{false} \\
\text{for each basic block} & \\
\text{apply equations updating IN or OUT} & \\
\text{if IN/OUT changes, set } \text{change} \text{ to true} \\
\text{end}
\end{align*}
\]

Consider: Does the order of blocks matter?
Recall: Liveness Analysis

<table>
<thead>
<tr>
<th>Block</th>
<th>LiveOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>{a, b, e}</td>
</tr>
<tr>
<td>B2</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td>B3</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td>B4</td>
<td>{a, b, c, d, e}</td>
</tr>
<tr>
<td>B5</td>
<td>{a, b, d, e}</td>
</tr>
<tr>
<td>B6</td>
<td>{}</td>
</tr>
</tbody>
</table>

```
B1
  d1: a := 1
  d2: b := 2

B2
  d3: c := a + b
  d4: d := c - a

B3
  d5: d := b * d

B4
  d6: d := a + b
  d7: e := e + 1

B5
  d8: b := a + b
  d9: e := c - a

B6
  d10: a := b * d
  d11: b := a - d
```
## Order of Dataflow Analysis

4 iterations to reach a fixed point using in-order CFG traversal

| Block | VarKill  | UEVar | SUCC | Iter 0 | Iter 1 | Iter 2 | Iter 3 | ...
|-------|----------|-------|------|--------|--------|--------|--------|------
| B1    | \{a,b\} | \{\}  | B2   | \{\}   | \{a,b\} | \{a,b\} | \{a,b,e\} | ... |
| B2    | \{c,d\} | \{a,b\} | B3, B5 | \{\}   | \{a,b,c,d\} | \{a,b,c,d,e\} | \{a,b,c,d,e\} | ... |
| B3    | \{d\}   | \{b,d\} | B4, B5 | \{\}   | \{a,b,c,e\} | \{a,b,c,e,d\} | \{a,b,c,d,e\} | ... |
| B4    | \{d,e\} | \{a,b,e\} | B3   | \{\}   | \{b,d,a,c,e\} | \{b,d,a,c,e\} | \{a,b,c,d,e\} | ... |
| B5    | \{b,e\} | \{a,b,c\} | B2, B6 | \{\}   | \{a,b,d\} | \{a,b,d,e\} | \{a,b,d,e\} | ... |
| B6    | \{a,b\} | \{b,d\} | -    | \{\}   | \{\}   | \{\}   | \{\}   | ... |
Order of Dataflow Analysis (2)

<table>
<thead>
<tr>
<th>Block</th>
<th>VarKill</th>
<th>UEVar</th>
<th>SUCC</th>
<th>Iter 0</th>
<th>Iter 1</th>
<th>Iter 2</th>
<th>Iter 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>B6</td>
<td>{a,b}</td>
<td>{b,d}</td>
<td>-</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>B5</td>
<td>{b,e}</td>
<td>{a,b,c}</td>
<td>B2,B6</td>
<td>{ }</td>
<td>{a,b,d}</td>
<td>{a,b,d,e}</td>
<td>{a,b,d,e}</td>
</tr>
<tr>
<td>B3</td>
<td>{d}</td>
<td>{b,d}</td>
<td>B4,B5</td>
<td>{ }</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
</tr>
<tr>
<td>B2</td>
<td>{c,d}</td>
<td>{a,b}</td>
<td>B3,B5</td>
<td>{ }</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
</tr>
<tr>
<td>B1</td>
<td>{a,b}</td>
<td>{ }</td>
<td>B2</td>
<td>{ }</td>
<td>{a,b,e}</td>
<td>{a,b,e}</td>
<td>{a,b,e}</td>
</tr>
<tr>
<td>B4</td>
<td>{d,e}</td>
<td>{a,b,e}</td>
<td>B3</td>
<td>{ }</td>
<td>{b,d,a,c,e}</td>
<td>{a,b,c,d,e}</td>
<td>{a,b,c,d,e}</td>
</tr>
</tbody>
</table>

3 iterations to reach a fixed point if we go in a different order

**N.B.** Correctness is unaffected, but optimization can go faster
Limitations of Dataflow Analysis

- **Correctness**
  - Iterative algorithm finds a fixed-point
    - Conservative guesses to maintain correctness
    - Not always the ideal solution for all problems of interest (e.g., constant propagation)

- **Complex data structures**
  - Arrays (usually represented as a single fact)
  - Pointers (aliasing, pointer arithmetic make things difficult)

- Often, we apply dataflow analysis to **scalar** problems
Redundancy elimination

▶ **Idea** In addition to propagating constant values, can we find entire expressions to propagate?

```cpp
foo (a : Int, b : Int, c : Int) : Object {
    let x : Int, y : Int in {
        x <- -(b*b) + 4 * a ;
        y <- -(b*b) + 4 * a * c;
        (* isn’t this the same as: *)
        y <- x * c;
    }
};
```
Redundancy elimination (2)

An expression \( \text{op} \ x \ y \) is *redundant* at a point \( p \) if it has already been computed and no intervening operations redefine \( x \) or \( y \)

\[
\begin{align*}
  m &= 2*y*z \\
  n &= 3*y*z \\
  o &= 2*y - z \\
  t0 &= 2*y \\
  t1 &= 3*y \\
  t2 &= 2*y \\
  t0 &= t0*z \\
  t1 &= t1*z \\
  o &= t2*z
\end{align*}
\]

redundant
Value Numbering for RE

- Goal: Group expressions together that provably have the same value

- Associate a unique value number with each distinct value created or used within a block
  - Two expressions have the same value number if and only if they are provably identical for all possible operands

- Propagate VNs around CFG to find opportunities to eliminate redundant code locally and globally
Example RE via VN

\[ a^3 = i^1 + 1^2 \]
\[ b = i + 1 \]
\[ i = j \]
\[ \text{if } i + 1 \text{ goto L1} \]

For variable a:
hash <+, VN(i), VN(1)> get a new number 3

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(1)+(2)</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
</tbody>
</table>

(#) means the entry associated with number #
Example RE via VN (2)

\[ a^3 = i^1 + 1^2 \]
\[ b^3 = i^1 + 1^2 \]
\[ i = j \]
\[ \text{if } i + 1 \text{ goto L1} \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(1)+(2)</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
</tr>
</tbody>
</table>

For variable b:
hash <+, VN(i), VN(1)>  
get an existing number 3
Example RE via VN (3)

\[ a^3 = i^1 + 1^2 \]
\[ b^3 = i^1 + 1^2 \]
\[ i^4 = j^4 \]

if \( i + 1 \) goto L1

<table>
<thead>
<tr>
<th></th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

(1)+(2) 3

<table>
<thead>
<tr>
<th></th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>4</td>
</tr>
</tbody>
</table>

VN(i) is changed to 4
Example RE via VN (4)

\[ a^3 = i^1 + 1^2 \]
\[ b^3 = i^1 + 1^2 \]
\[ i^4 = j^4 \]
\[ i^4 + 1^2 \text{ goto L1} \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>(1)+(2)</td>
<td>3</td>
</tr>
<tr>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>4</td>
</tr>
<tr>
<td>(2)+(4)</td>
<td>5</td>
</tr>
</tbody>
</table>
Example RE via VN (5)

\[
\begin{align*}
a^3 &= i^1 + 1^2 \\
b^3 &= i^1 + 1^2 \\
i^4 &= j^4 \\
\text{if } i^4 + 1^2 \text{ goto L1}
\end{align*}
\]

\[
\begin{align*}
a &= i + 1 \\
b &= a \\
i &= j \\
\text{if } i + 1 \text{ goto L1}
\end{align*}
\]

a and b are given the same numbering, but not the condition expression for the if statement
RE via Dataflow Analysis

- Value numbering may miss opportunities (e.g., variable $i$ in that last example)

- What we really need is to propagate the expressions that are available at a given point

- We have seen a simple example developing intuition via value numbering, but can we fit this problem into a general dataflow analysis?
RE via Dataflow Analysis (2)

- An expression $e$ if **defined** at point $p$ if its value is computed at $p$
  - $p$ is called a definition site for $e$
- An expression $e$ is **killed** at point $p$ if one or more of its operands is defined at $p$
  - $p$ is called a kill site for $e$
- An expression $e$ is **available** at point $p$ if every path leading to $p$ contains a definition of $e$, and $e$ is not killed between that definition and $p$
- An expression $\text{op } x \ y$ is **redundant** at a point $p$ if is has already been computed and no intervening operations redefine $x$ or $y$
RE via Dataflow Analysis Example

Definition site

Since $a + b$ is available here, $\Rightarrow$ redundant!
RE via Dataflow Analysis Example (2)

Definition site

c = a + b

Kill site

d = a * c

Candidates:

f[i] = a + b

Not available ➡ Not redundant

c = c * 2

if c > d

g = a * c

i = 1

i = i + 1

if i > 10

g = d * d

da + b

a * c
d * d

c * 2
i + 1
Redundancy elimination remarks

- General dataflow analysis for redundancy elimination involves available expression analysis:
  - \( \text{DEExp}(b) \): Downward Exposed expressions
    - \( e \in \text{DEExpr}(b) \) if \( b \) evaluates \( e \) and none of \( e \)'s operands is re-defined after that evaluation
    - In contrast to \( \text{UEExpr}(b) \), upward-exposed expressions
  - \( \text{ExprKill}(b) \) – expressions killed by definitions in the block

- Dataflow analyses can be boiled down to defining IN and OUT sets in terms of \( \text{UEExpr} \), \( \text{DEExpr} \), etc. for each block
  - Then you move IN and OUT sets around the CFG according to your dataflow rules
# Dataflow Analyses

<table>
<thead>
<tr>
<th>Domain</th>
<th>AVAIL</th>
<th>REACHES</th>
<th>LIVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction</td>
<td>Set of expressions</td>
<td>set of definitions</td>
<td>set of variables</td>
</tr>
<tr>
<td>Forward</td>
<td>Forward</td>
<td>Backward</td>
<td></td>
</tr>
<tr>
<td>Across blocks</td>
<td>$AVAIL(B) = \bigcap_{x \in \text{PRED}(B)} OUT(X)$</td>
<td>$REACH(B) = \bigcup_{x \in \text{PRED}(B)} OUT(X)$</td>
<td>$LIVE(B) = \bigcup_{x \in \text{SUCC}(B)} IN(X)$</td>
</tr>
<tr>
<td>Initialize</td>
<td>$AVAIL(B) = {E}$</td>
<td>$REACH(B) = {}$</td>
<td>$LIVE(B) = {}$</td>
</tr>
</tbody>
</table>

Dataflow Analysis can be generalized into a format like this

All that’s left is a way to compute information within each block (easy!)