Compiler Development (CMPSC 401)
Code Optimization

Janyl Jumadinova

April 15, 2019
Goal:
Optimize generated code by exploiting machine-dependent properties not visible at the IR level.
Goal:
Optimize generated code by exploiting machine-dependent properties not visible at the IR level.
Goal:
Optimize generated code by exploiting machine-dependent properties not visible at the IR level.

- Critical step in most compilers, but often very messy.
- Techniques developed for one machine may be completely useless on another.
- Techniques developed for one language may be completely useless with another.
ARM vs. Intel’s x86

- ARM has an advantage in terms of power consumption, making it attractive for all sorts of battery operated devices.
Address space: $2^{32}$

Data types:
- 8, 16, 32, 64 bit int, signed and unsigned
- 32 and 64-bit floating point
- Binary coded decimal
- 64, 128, 256 bit vectors of integers/floats
x86 Registers Overview

16-bit integer registers

- General purpose (with exceptions): AX, BX, CX, DX
- Pointer registers: SP (Stack pointer), BP (Base Pointer)
- For array indexing: DI, SI
- Segment registers: CS, DS, SS, ES (legacy)
- FLAGS register to store flags, e.g. CF, OF, ZF
- Instruction Pointer: IP
For instructions see Intel Software developers manual

“The x86 isn’t all that complex... it just doesn’t make a lot of sense” Mike Johnson, AMD, 1994
Goal:
Optimize generated code by exploiting machine-dependent properties not visible at the IR level.
add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t5, $t3, $t4  # $t5 = $t3 + $t4
add $t8, $t6, $t7  # $t8 = $t6 + $t7
Processor Pipelines

```
add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t5, $t3, $t4  # $t5 = $t3 + $t4
add $t8, $t6, $t7  # $t8 = $t6 + $t7
```
Processor Pipelines

Instruction Decoder

Register File

ALU

Read Port

Write Port

\[
\begin{align*}
\text{add } & \text{ $t2, t0, t1} & \# t2 = t0 + t1 \\
\text{add } & \text{ $t5, t3, t4} & \# t5 = t3 + t4 \\
\text{add } & \text{ $t8, t6, t7} & \# t8 = t6 + t7
\end{align*}
\]
Processor Pipelines

Instruction Decoder

Register File

ALU

Read Port

Write Port

add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t5, $t3, $t4  # $t5 = $t3 + $t4
add $t8, $t6, $t7  # $t8 = $t6 + $t7
Processor Pipelines

This value isn't ready yet!

Instruction Decoder

Register File

Read Port

Write Port

ALU

ID RR ALU RW

```
add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t4, $t3, $t2  # $t5 = $t3 + $t2
add $t7, $t5, $t6  # $t7 = $t5 + $t6
add $t0, $t0, $t7  # $t0 = $t0 + $t7
```
Processor Pipelines

Stall pipeline until value is ready

<table>
<thead>
<tr>
<th>Instruction Decoder</th>
<th>Register File</th>
<th>ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Port</td>
<td>Write Port</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>RR</th>
<th>ALU</th>
<th>RW</th>
</tr>
</thead>
</table>

```
add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t4, $t3, $t2  # $t5 = $t3 + $t2
add $t7, $t5, $t6  # $t7 = $t5 + $t6
add $t0, $t0, $t7  # $t0 = $t0 + $t7
```
Processor Pipelines

\[
\begin{align*}
\text{add } s & \text{t2, } s & \text{t0, } s & \text{t1} & \# s & \text{t2} &= s & \text{t0} + s & \text{t1} \\
\text{add } s & \text{t4, } s & \text{t3, } s & \text{t2} & \# s & \text{t5} &= s & \text{t3} + s & \text{t2} \\
\text{add } s & \text{t7, } s & \text{t5, } s & \text{t6} & \# s & \text{t7} &= s & \text{t5} + s & \text{t6} \\
\text{add } s & \text{t0, } s & \text{t0, } s & \text{t7} & \# s & \text{t0} &= s & \text{t0} + s & \text{t7}
\end{align*}
\]
Processor Pipelines

**Instruction Decoder**

**Register File**

**ALU**

Read Port  
Write Port

```
add $t2, $t0, $t1  # $t2 = $t0 + $t1
add $t7, $t5, $t6  # $t7 = $t5 + $t6
add $t4, $t3, $t2  # $t5 = $t3 + $t2
add $t0, $t0, $t7  # $t0 = $t0 + $t7
```

Two clock cycles faster!
Because of processor pipelining, the order in which instructions are executed can impact performance.

**Instruction scheduling** is the reordering or insertion of machine instructions to increase performance.

All good optimizing compilers have some sort of instruction scheduling support.
A data dependency in machine code is a set of instructions whose behavior depends on one another.

Intuitively, a set of instructions that cannot be reordered around each other.

Three types of data dependencies:

- **Read-after-Write (RAW)**
  - \( x = \ldots \)
  - \( \ldots = x \)
  - \( x = \ldots \)

- **Write-after-Read (WAR)**
  - \( \ldots = x \)
  - \( x = \ldots \)

- **Write-after-Write (WAW)**
  - \( x = \ldots \)
  - \( x = \ldots \)
Finding Data Dependencies

t0 = t1 + t2

\[
t1 = t0 + t1 \\
t3 = t2 + t4 \\
t0 = t1 + t2 \\
t5 = t3 + t4 \\
t6 = t2 + t3
\]
Finding Data Dependencies

t0 = t1 + t2

t1 = t0 + t1

t3 = t2 + t4

t0 = t1 + t2

t5 = t3 + t4

t6 = t2 + t7
Finding Data Dependencies

\[ t_3 = t_2 + t_4 \]
\[ t_5 = t_3 + t_4 \]
\[ t_0 = t_1 + t_2 \]
\[ t_1 = t_0 + t_1 \]
\[ t_6 = t_2 + t_7 \]
\[ t_0 = t_1 + t_2 \]
The graph of the data dependencies in a basic block is called the data dependency graph.
The graph of the data dependencies in a basic block is called the data dependency graph.

Always a directed acyclic graph:

- **Directed**: One instruction depends on the other.
- **Acyclic**: No circular dependencies allowed.

Can schedule instructions in a basic block in any order as long we never schedule a node before all its parents.
The graph of the data dependencies in a basic block is called the data dependency graph.

Always a directed acyclic graph:

- **Directed**: One instruction depends on the other.
- **Acyclic**: No circular dependencies allowed.

Can schedule instructions in a basic block in any order as long we never schedule a node before all its parents.

**Idea**: Do a topological sort of the data dependency graph and output instructions in that order.
Instruction Scheduling

\[ t_3 = t_2 + t_4 \]
\[ t_5 = t_3 + t_4 \]
\[ t_0 = t_1 + t_2 \]
\[ t_1 = t_0 + t_1 \]
\[ t_6 = t_2 + t_7 \]
\[ t_0 = t_1 + t_2 \]
Instruction Scheduling

t3 = t2 + t4

\[ t5 = t3 + t4 \]

\[ t0 = t1 + t2 \]

\[ t6 = t2 + t7 \]

\[ t0 = t1 + t2 \]
Instruction Scheduling

t3 = t2 + t4

\[ t5 = t3 + t4 \]

\[ t0 = t1 + t2 \]

\[ t6 = t2 + t7 \]

\[ t0 = t1 + t2 \]

\[ t0 = t1 + t2 \]

\[ t0 = t1 + t2 \]
Small Problem

- There can be many valid topological orderings of a data dependency graph.
- How do we pick one that works well with the pipeline?
- In general, finding the fastest instruction schedule is known to be **NP-hard**.
- Heuristics are used in practice:
There can be many valid topological orderings of a data dependency graph.

How do we pick one that works well with the pipeline?

In general, finding the fastest instruction schedule is known to be NP-hard.

Heuristics are used in practice:

- Schedule instructions that can run to completion without interference before instructions that cause interference.
- Schedule instructions with more dependants before instructions with fewer dependants.
Modern optimizing compilers can do far more aggressive scheduling to obtain impressive performance gains.

**Loop unrolling**
- Expand out several loop iterations at once.
- Use previous algorithm to schedule instructions more intelligently.

Can find pipelining-level parallelism across loop iterations.
Modern optimizing compilers can do far more aggressive scheduling to obtain impressive performance gains.

**Loop unrolling**
- Expand out several loop iterations at once.
- Use previous algorithm to schedule instructions more intelligently.

Can find pipelining-level parallelism across loop iterations.

**Software pipelining**
- Loop unrolling on steroids; can convert loops using tens of cycles into loops averaging two or three cycles.
Because computers use different types of memory, there are a variety of memory caches in the machine.

- Caches are designed to anticipate common use patterns.
- Compilers often have to rewrite code to take maximal advantage of these designs.
Empirically, many programs exhibit temporal locality and spatial locality.
Empirically, many programs exhibit temporal locality and spatial locality.

**Temporal locality**: Memory read recently is likely to be read again.

**Spatial locality**: Memory read recently will likely have nearby objects read as well.

Most memory caches are designed to exploit temporal and spatial locality by
- Holding recently-used memory addresses in cache.
- Loading nearby memory addresses into cache.
Memory Caches

\[
\begin{align*}
\text{arr[0]} & = 5; \\
\text{arr[2]} & = 6; \\
\text{arr[10]} & = 13; \\
\text{arr[1]} & = 4;
\end{align*}
\]
\[
\begin{align*}
\text{arr}[0] & = 5; \\
\text{arr}[2] & = 6; \\
\text{arr}[10] & = 13; \\
\text{arr}[1] & = 4; \\
\end{align*}
\]
arr[0] = 5;
arr[2] = 6;
arr[10] = 13;
arr[1] = 4;
Memory Caches

```python
arr[0] = 5;
arr[2] = 6;
arr[10] = 13;
arr[1] = 4;
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[0]</td>
<td>5</td>
</tr>
<tr>
<td>arr[1]</td>
<td>0</td>
</tr>
<tr>
<td>arr[2]</td>
<td>6</td>
</tr>
<tr>
<td>arr[3]</td>
<td>0</td>
</tr>
<tr>
<td>arr[4]</td>
<td>0</td>
</tr>
<tr>
<td>arr[5]</td>
<td>0</td>
</tr>
<tr>
<td>arr[6]</td>
<td>0</td>
</tr>
<tr>
<td>arr[7]</td>
<td>0</td>
</tr>
<tr>
<td>arr[8]</td>
<td>0</td>
</tr>
<tr>
<td>arr[9]</td>
<td>0</td>
</tr>
<tr>
<td>arr[10]</td>
<td>0</td>
</tr>
<tr>
<td>arr[11]</td>
<td>0</td>
</tr>
</tbody>
</table>

Memory Cache

```plaintext

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>arr[8]</td>
<td>0</td>
</tr>
<tr>
<td>arr[9]</td>
<td>0</td>
</tr>
<tr>
<td>arr[10]</td>
<td>0</td>
</tr>
<tr>
<td>arr[11]</td>
<td>0</td>
</tr>
</tbody>
</table>
```
Memory Caches

```plaintext
arr[0] = 5;
arr[2] = 6;
arr[10] = 13;
arr[1] = 4;
```

![Memory Cache Diagram]
Memory Caches

\[
\begin{align*}
\text{arr[0]} & = 5; \\
\text{arr[2]} & = 6; \\
\text{arr[10]} & = 13; \\
\text{arr[1]} & = 4;
\end{align*}
\]
Memory Caches

```
arr[0] = 5;
arr[2] = 6;
arr[10] = 13;
arr[1] = 4;
```
Improving Locality

- Programmers frequently write code without understanding the locality implications.
- Languages don’t expose low-level memory details.
- Some compilers are capable of rewriting code to take advantage of locality.
Improving Locality

- Programmers frequently write code without understanding the locality implications.
- Languages don’t expose low-level memory details.
- Some compilers are capable of rewriting code to take advantage of locality.
  - Loop reordering.
  - Structure peeling.
int[][] array;
for (j = 0; j < 4; j = j + 1)
    for (i = 0; i < 4; i = i + 1)
        array[i][j] = 0;

| 00 | 01 | 02 | 03 | 10 | 11 | 12 | 13 | 20 | 21 | 22 | 23 | 30 | 31 | 32 | 33 |

Cache
int[][] array;
for (j = 0; j < 4; j = j + 1)
    for (i = 0; i < 4; i = i + 1)
        array[i][j] = 0;

00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

Cache
int[][] array;
for (i = 0; i < 4; i = i + 1)
    for (j = 0; j < 4; j = j + 1)
        array[i][j] = 0;

| 00 | 01 | 02 | 03 | 10 | 11 | 12 | 13 | 20 | 21 | 22 | 23 | 30 | 31 | 32 | 33 |

Cache
int[][] array;
for (i = 0; i < 4; i = i + 1)
    for (j = 0; j < 4; j = j + 1)
        array[i][j] = 0;

00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

Cache
00 01 02 03
int[][] array;
for (i = 0; i < 4; i = i + 1)
    for (j = 0; j < 4; j = j + 1)
        array[i][j] = 0;

00 01 02 03 10 11 12 13 20 21 22 23 30 31 32 33

Cache
int[][] array;
for (i = 0; i < 4; i = i + 1)
    for (j = 0; j < 4; j = j + 1)
        array[i][j] = 0;
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];

    /* ... initialize the points ... */

    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];
    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];

    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];

    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}

Memory Cache

pts[2].x  
pts[2].y  
pts[3].x  
pts[3].y  
pts[4].x  
pts[4].y  
pts[5].x  
pts[5].y  
...
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];
    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];

    /* ... initialize the points ... */

    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];
    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];
    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
Structure Peeling

class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];
    /* ... initialize the points ... */
    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
```java
class Point2D {
    int x;
    int y;
}

void MyFunction() {
    Point2D[] pts = new Point2D[1024];
    /* ... initialize the points ... */

    int maxX = 0, maxY = 0;
    for (int i = 0; i < 512; ++i)
        maxX = max(pts[i].x, maxX);
    for (int i = 512; i < 1024; ++i)
        maxY = max(pts[i].y, maxY);
}
```
Instruction scheduling optimizations try to take advantage of the processor pipeline.

Locality optimizations try to take advantage of cache behavior.

Parallelism optimizations try to take advantage of multi-core machines.

There are many more optimizations out there!
Where we have been

Source Code

Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

Machine Code
Where we have been

- Regular Expressions
- Finite Automata
- Subset Construction
- flex

<table>
<thead>
<tr>
<th>Lexical Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Analysis</td>
</tr>
<tr>
<td>Semantic Analysis</td>
</tr>
<tr>
<td>IR Generation</td>
</tr>
<tr>
<td>IR Optimization</td>
</tr>
<tr>
<td>Code Generation</td>
</tr>
<tr>
<td>Optimization</td>
</tr>
</tbody>
</table>
Where we have been

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

• Context-Free Grammars
• Parse Trees
• ASTs
• Leftmost DFS
• LL(1)
• Handles
• LR(0)
• LR(1)
• LALR(1)
• bison
Where we have been

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

- Scope-Checking
- Spaghetti Stacks
- Function Overloading
- Type Systems
- Well-Formedness
- Null and Error Types
Where we have been

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

Graphical Representations
- Control Flow Graph
- Dependence Graph
- Concrete/Abstract Syntax Trees (ASTs)

Linear Representations
- Stack based
- Three-Address Code
Where we have been

<table>
<thead>
<tr>
<th>Lexical Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Analysis</td>
</tr>
<tr>
<td>Semantic Analysis</td>
</tr>
<tr>
<td>IR Generation</td>
</tr>
<tr>
<td><strong>IR Optimization</strong></td>
</tr>
<tr>
<td>Code Generation</td>
</tr>
<tr>
<td>Optimization</td>
</tr>
</tbody>
</table>

- Basic Blocks
- Control-Flow Graphs
- Common Subexpression Elimination
- Copy Propagation
- Dead Code Elimination
- Arithmetic Simplification
- Constant Folding
- Global Constant Propagation
- Partial Redundancy Elimination
Where we have been

<table>
<thead>
<tr>
<th>Lexical Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax Analysis</td>
</tr>
<tr>
<td>Semantic Analysis</td>
</tr>
<tr>
<td>IR Generation</td>
</tr>
<tr>
<td>IR Optimization</td>
</tr>
<tr>
<td><strong>Code Generation</strong></td>
</tr>
<tr>
<td>Optimization</td>
</tr>
</tbody>
</table>

- ARM
Where we have been

- Instruction Scheduling
- Loop Reordering
- Structure Peeling
- Automatic Parallelization
Why Study Compilers?

- Build a large, ambitious software system.
- See theory come to life.
- Learn how programming languages work.
- Learn tradeoffs in language design.