The Problem

After compilation, the code still has some flaws

• Scheduling & allocation really are NP-Complete
• Optimizer may not implement every needed transformation

Curing the problem

• More work on scheduling and allocation
• Implement more optimizations

—or—

• Optimize after compilation
  > Peephole optimization
  > Link-time optimization
Post-compilation Optimization

Field has attracted much attention lately

• New & interesting opportunities exist at link time & at run time
  > Interprocedural analysis & optimization
  > Resolved references & data-structure sizes
  > Code that is out of scope at other times, such as linkage code
  > Optimize object-only distributions of libraries & applications

• Hard problems in this arena
  > Reconstructing control flow

Unravelling Control-flow (TI TMS320C6x)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>B .S1 LOOP</td>
<td>branch to loop</td>
</tr>
<tr>
<td>B .S1 LOOP</td>
<td>branch to loop</td>
</tr>
<tr>
<td>B .S1 LOOP</td>
<td>branch to loop</td>
</tr>
<tr>
<td>B .S1 LOOP</td>
<td>branch to loop</td>
</tr>
<tr>
<td>ZERO .L1 A2</td>
<td>zero A side product</td>
</tr>
<tr>
<td>ZERO .L2 B2</td>
<td>zero B side product</td>
</tr>
<tr>
<td>ZERO .L1 A3</td>
<td>zero A side accumulator</td>
</tr>
<tr>
<td>ZERO .L2 B3</td>
<td>zero B side accumulator</td>
</tr>
<tr>
<td>ZERO .D1 A1</td>
<td>zero A side load value</td>
</tr>
<tr>
<td>ZERO .D2 B1</td>
<td>zero B side load value</td>
</tr>
<tr>
<td>LDW .D1 *A4++, A1</td>
<td>load a[i] &amp; a[i+1]</td>
</tr>
<tr>
<td>LDW .D2 *B4++, B1</td>
<td>load a[i] &amp; a[i+1]</td>
</tr>
<tr>
<td>MPY .M1X A1, B1, A2</td>
<td>load b[i] &amp; b[i+1]</td>
</tr>
<tr>
<td>MPYH .M2X A1, B1, B2</td>
<td>a[i] * b[i]</td>
</tr>
<tr>
<td>ADD .L1 A2, A3, A3</td>
<td>a[i+1] * b[i+1]</td>
</tr>
<tr>
<td>ADD .L2 B2, B3, B3</td>
<td>ca += a[i] * b[i]</td>
</tr>
<tr>
<td>SUB .S2 B0, 1, B0</td>
<td>decrement loop counter</td>
</tr>
<tr>
<td>B .S1 LOOP</td>
<td>branch to loop</td>
</tr>
<tr>
<td>ADD .L1X A3, B3, A3</td>
<td>c = ca + cb</td>
</tr>
</tbody>
</table>

Stuff 4 branches into the pipe
Set up the loop + 1 more branch
Single cycle loop ending with another branch
Unravelling Control-flow  (TI TMS320C6x)

```
B .S1 LOOP ; branch to loop
B .S1 LOOP ; branch to loop
B .S1 LOOP ; branch to loop
B .S1 LOOP ; branch to loop
|| ZERO .L1 A2 ; zero A side product
|| ZERO .L2 B2 ; zero B side product
From this, the post-optimization translator is
supposed to recover a simple source code
loop ...
|| ZERO .D1 A1 ; zero A side load value
|| ZERO .D2 B1 ; zero B side load value
LOOP: LDW .D1 *A4++, A1 ; load a[i] & a[i+1]
|| LDW .D2 *B4++, B1 ; load b[i] & b[i+1]
|| MPY .M1X A1, B1, A2 ; load b[i] & b[i+1]
|| MPYH .M2X A1, B1, B2 ; a[i] * b[i]
|| ADD .L1 A2, A3, A3 ; a[i+1] * b[i+1]
|| ADD .L2 B2, B3, B3 ; ca += a[i] * b[i]
|| [B0] SUB .S2 B0, 1, B0 ; decrement loop counter
|| [B0] B .S1 LOOP ; branch to loop
ADD .L1X A3, B3, A3 ; c = ca + cb
```

From this, the post-optimization translator is supposed to recover a simple source code loop ...

```
|| ZERO .D1 A1 ; zero A side load value
|| ZERO .D2 B1 ; zero B side load value
LOOP: LDW .D1 *A4++, A1 ; load a[i] & a[i+1]
|| LDW .D2 *B4++, B1 ; load b[i] & b[i+1]
|| MPY .M1X A1, B1, A2 ; load b[i] & b[i+1]
|| MPYH .M2X A1, B1, B2 ; a[i] * b[i]
|| ADD .L1 A2, A3, A3 ; a[i+1] * b[i+1]
|| ADD .L2 B2, B3, B3 ; ca += a[i] * b[i]
|| [B0] SUB .S2 B0, 1, B0 ; decrement loop counter
|| [B0] B .S1 LOOP ; branch to loop
ADD .L1X A3, B3, A3 ; c = ca + cb
```

Post-compilation Optimization

Field has attracted much attention lately

- New & interesting opportunities exist at link time & at run time
  - Interprocedural analysis & optimization
  - Resolved references & data-structure sizes
  - Code that is out of scope at other times, such as linkage code
  - Optimize object-only distributions of libraries & applications

- Hard problems in this arena
  - Reconstructing control flow
  - Reconstructing type & dimension information
  - Analyzing pointers & addresses

While whole-program techniques are expensive, at link-time the compiler must traverse all that code anyway.
Potential Post-compilation Optimizations

A mix of old and new techniques

- Peephole optimization
- Code positioning (& branch optimizations)
- Sinking (cross jumping) & hoisting
- Procedure abstraction (for space)
- Register reallocation (scavenging, Belady, coloring)
- Bit-transition reduction

- Dead & Clean
- Constant propagation
- Pointer disambiguation & register promotion

Peephole Optimization

The Basic Idea

- Discover local improvements by looking at a window on the code
  > A tiny window is good enough — a peephole
- Slide the peephole over the code and examine the contents
  > Pattern match with a limited set of patterns

Examples

\[
\begin{align*}
\text{storeAl } r_4 & \Rightarrow r_{0,8} \\
\text{loadAl } r_{0,8} & \Rightarrow r_{15} \\
\text{addI } r_2,0 & \Rightarrow r_7 \\
\text{mult } r_4,r_7 & \Rightarrow r_{10} \\
\text{jumpI } l_{10} & \Rightarrow l_{10} \\
\text{jumpI } l_{11} & \Rightarrow l_{11}
\end{align*}
\]

\[
\begin{align*}
\text{storeAl } r_1 & \Rightarrow r_{0,8} \\
\text{i2i } r_1 & \Rightarrow r_{15} \\
\text{mult } r_4,r_2 & \Rightarrow r_{10} \\
\text{jumpI } l_{10} & \Rightarrow l_{11} \\
\text{jumpI } l_{11} & \Rightarrow l_{11}
\end{align*}
\]

Less likely as a local sequence, but other opts can create it...


**Peephole Optimization**

**Early Peephole Optimizers** (McKeeman)

- Used limited set of hand-coded patterns
- Matched with exhaustive search
- Small window, small pattern set \(\Rightarrow\) quick execution
- They proved effective at cleaning up the rough edges
  - Code generation is inherently local
  - Boundaries between local regions are trouble spots

Improvements in code generation, optimization, & architecture should have let these fade into obscurity
  - Much better allocation & scheduling today than in 1965
  - But, we have much more complex architectures

**Modern Peephole Optimizers** (Davidson, Fraser)

- Larger, more complex ISAs \(\Rightarrow\) larger pattern sets
- This has produced a more systematic approach

![Diagram](asm-expander-simplifiermatcher.png)

**Expander**

- Operation-by-operation expansion into LLIR
- Needs no context
- Captures full effect of an operation
Peephole Optimization

Modern Peephole Optimizers (Davidson, Fraser)

- Larger, more complex ISAs ⇒ larger pattern sets
- This has produced a more systematic approach

Simplifier

- Single pass over LLIR, moving the peephole
- Forward substitution, algebraic simplification, constant folding, & eliminating useless effects \(\text{(must know what is dead)}\)
- Eliminate as many LLIR operations as possible

Matcher

- Starts with reduced LLIR program
- Compares LLIR from peephole against pattern library
- Selects one or more ASM patterns that “cover” the LLIR
  - Capture all of its effects
  - May create new useless effects \(\text{(setting the } cc\text{)}\)
**Finding Dead Effects**

The simplifier must know what is useless \(i.e.,\) dead

- Expander works in a context-independent fashion
- It can process the operations in any order
  - Use a backward walk and compute local LIVE information
  - Tag each operation with a list of useless values
- What about non-local effects?
  - Most useless effects are local — DEF & USE in same block
  - It can be conservative & assume LIVE until proven dead

\[
\begin{align*}
\text{ASM} & \quad \text{LLIR} & \quad \text{LLIR} & \quad \text{ASM} \\
\text{mult } r_5, r_9 \Rightarrow r_{12} & \quad \Rightarrow r_{12} & \quad \Rightarrow r_{12} & \quad \Rightarrow r_{12} \\
\text{add } r_{12}, r_{17} \Rightarrow r_{13} & \quad \Rightarrow r_{13} & \quad \Rightarrow r_{13} & \quad \Rightarrow r_{13} \\
\text{cc} & \quad \Rightarrow f(r_5 + r_9) & \quad \Rightarrow f(r_{12} + r_{17}) & \quad \Rightarrow f(r_{12} + r_{17}) \\
\text{match} & \quad \Rightarrow madd r_5, r_9, r_{17} \Rightarrow r_{13} & \quad \Rightarrow r_{13} & \quad \Rightarrow r_{13} \\
\end{align*}
\]

This effect would prevent multiply-add from matching

---

**Peephole Optimization**

Can use it to perform instruction selection

- Key issue in selection is effective pattern matching

\[
\begin{align*}
\text{ASM} & \quad \text{LLIR} & \quad \text{LLIR} & \quad \text{ASM} \\
\text{Expander} & \quad \Rightarrow \text{Simplifier} & \quad \Rightarrow \text{Matcher} & \quad \Rightarrow \text{ASM} \\
\end{align*}
\]

Using peephole system for instruction selection

- Have front-end generate LLIR directly
- Eliminates need for the Expander
- Keep Simplifier and Matcher
  - Add a simple register assigner, follow with real allocation

This basic scheme is used in GCC
Peephole-Based Instruction Selection

Basic Structure of Compilers like GCC

- Uses RTL as its IR (very low level, register-transfer language)
- Numerous optimization passes
- Quick translation into RTL limits what optimizer can do ...
- Matcher generated from specs (hard-coded tree-pattern matcher)

An Example

Original Code
\[ w \leftarrow x - 2 \times y \]

Compiler's IR

<table>
<thead>
<tr>
<th>Op</th>
<th>Arg 1</th>
<th>Arg 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>2</td>
<td>y</td>
<td>t1</td>
</tr>
<tr>
<td>-</td>
<td>x</td>
<td>t1</td>
<td>w</td>
</tr>
</tbody>
</table>

Translation
\[ r_{10} \leftarrow 2 \\
 r_{11} \leftarrow @y \\
 r_{12} \leftarrow r_0 + r_{11} \\
 r_{13} \leftarrow M(r_{12}) \\
 r_{14} \leftarrow r_{10} \times r_{13} \\
 r_{15} \leftarrow @x \\
 r_{16} \leftarrow r_0 + r_{15} \\
 r_{17} \leftarrow M(r_{16}) \\
 r_{18} \leftarrow r_{17} \times r_{14} \\
 r_{19} \leftarrow @w \\
 r_{20} \leftarrow r_0 + r_{19} \\
 M(r_{20}) \leftarrow r_{18} \]
### Simplification with a three operation window

**Original Code**

Suppose we have the following code:

\[
\begin{align*}
  r_{10} &\leftarrow 2 \\
  r_{11} &\leftarrow @y \\
  r_{12} &\leftarrow r_0 + r_{11} \\
  r_{13} &\leftarrow r_0 + \times r_{12} \\
  r_{14} &\leftarrow 2 \times r_{13} \\
  r_{15} &\leftarrow @x \\
  r_{16} &\leftarrow r_0 + r_{15} \\
  r_{17} &\leftarrow M(r_{16}) \\
  r_{18} &\leftarrow r_{17} - r_{14} \\
  r_{19} &\leftarrow @w \\
  r_{20} &\leftarrow r_0 + r_{19} \\
  M(r_{20}) &\leftarrow r_{18}
\end{align*}
\]

No further improvement is found.

**Simplified Code**

\[
\begin{align*}
  r_{10} &\leftarrow 2 \\
  r_{11} &\leftarrow @y \\
  r_{12} &\leftarrow r_0 + r_{11} \\
  r_{13} &\leftarrow M(r_{12}) \\
  r_{14} &\leftarrow 2 \times r_{13} \\
  r_{15} &\leftarrow @x \\
  r_{16} &\leftarrow r_0 + r_{15} \\
  r_{17} &\leftarrow M(r_{16}) \\
  r_{18} &\leftarrow r_{17} - r_{14} \\
  r_{19} &\leftarrow @w \\
  r_{20} &\leftarrow r_0 + r_{19} \\
  M(r_{20}) &\leftarrow r_{18}
\end{align*}
\]

Simplification shrinks the code significantly.

**Example, Continued**

Simplification shrinks the code significantly by taking 5 operations instead of 12 and using 4 registers instead of 11.
Other Considerations

Control-flow operations

• Can clear simplifier’s window at branch or label
• More aggressive approach: combine across branches
  > Must account for effects on all paths
  > Not clear that this pays off ….
• Same considerations arise with predication

Physical versus logical windows

• Can run optimizer over a logical window
  > $k$ operations connected by DEF-USE chains
• Expander can link DEFs & USEs
• Logical windows (within block) improve effectiveness

Davidson & Fraser report 30% faster & 20% fewer ops with local logical window.

Peephole Optimization

So, …

• Peephole optimization remains viable
  > Post allocation improvements
  > Cleans up rough edges
• Peephole technology works for selection
  > Description driven matchers
    (hard coded or LR(1))
  > Used in several important systems
• Simplification pays off late in process
  > Low-level substitution, identities, folding, & dead effects

All of this will work equally well in binary-to-binary translation
Other Post-Compilation Techniques

What else makes sense to do after compilation?

✓ • Profile-guided code positioning
   > Allocation intact, schedule intact

✓ • Cross-jumping
   > Allocation intact, schedule changed

✓ • Hoisting
   > Changes allocation & schedule, needs data-flow analysis

✓ • Procedure abstraction
   > Changes allocation & schedule, really needs an allocator

✓ • Register scavenging
   > Changes allocation & schedule, purely local transformation

✓ • Bit-transition reduction
   > Schedule & allocation intact, assignment changed

Harder problems:
1. Rescheduling
2. Operation placement
3. Vectorization

Register Scavenging

Simple idea

• Global allocation does a good job on the big picture items
• Leaves behind blocks where some registers are unused

Let’s scavenge those unused registers

• Compute LIVE information
• Walk each block to find underallocated region
  > Find spilled local subranges
  > Opportunistically promote them to registers

A note of realism:

• Opportunities exist, but this is a 1% to 2% improvement
**Bit-transition Reduction**

Inter-operation bit-transitions relate to power consumption

- Large fraction of CMOS power is spent switching states
- Same op on same functional unit costs less power
  - All other things being equal

Simple idea

- Reassign registers to minimize interoperation bit transitions
- Build some sort of weighted graph
- Use a greedy algorithm to pick names by distance
- Should reduce power consumption in fetch & decode hardware

Waterman’s MS thesis (*in preparation*)

---

**Bit-transition Reduction**

Other transformations

- Swap operands on commutative operators
  - More complex than it sounds
  - Shoot for zero-transition pairs

- Swap operations within “fetch packets”
  - Works for superscalar, not VLIW

- Consider bit transitions in scheduling
  - Same ops to same functional unit
  - Nearby (Hamming distance) ops next, and so on ...

- Factor bit transitions into instruction selection
  - Maybe use a BURS model with dynamic costs

Again, most of this fits into a post-compilation framework…..