LLVM Register Allocation

Evan Cheng
Apple Inc.
August 1, 2008
LLVM Register Allocation

- Motivation
- Overview
- Optimizations
- Future Work
Isn’t It Done?

• Code generator does a reasonable job
  – LLVM code generator has proven to be quite capable
  – Roughly ~5% better than GCC 4.2 on x86 SPEC
  – About the same as GCC on x86-64
  – Even better on codecs

• But...
Really, Why Do We Care?

- Squeeze out that last few percentages of performance
- Fix the pathological cases
- Improve compile time for JIT and static codegen
- Enable more aggressive optimizations
LLVM Design Philosophy

- Each optimization pass should be as aggressive as possible
- Later passes must do *the right thing* to avoid pessimization
- Earlier optimization passes may increase register pressure
- Register allocation must be able to deal with the increased register pressure
Example: Machine LICM

loop_preheader:
brcc bb1

bb1:
v1 = VSET0  // xor xmm0,xmm0
...  
  = v1
...  
brcc bb1

This must be good, right?
Example: Machine LICM cont.

• Not necessarily!

It increases register pressure so v1 may be spilled

loop_preheader:

v1 = VSET0  // xor xmm0,xmm0
store v1, [fi#1]
brcc bb1

bb1:

... v1.1 = load [fi#1]
    = v1.1
...
brcc bb1
LLVM Register Allocation

- Motivation
- Overview
- Optimizations
- Future Work
Design of the Register Allocator

PHI Elimination → Two-Address → Coalescing

Linear Scan Allocator ↔ Spiller

Rewriter
**PHI Elimination**

Move code out of SSA form and eliminate PHI instructions

BB1:
\[ v1 = \]
\[ v5 = v1 \]

BB2:
\[ v2 = \]
\[ v5 = v2 \]

BB3:
\[ v3 = \]
\[ v5 = v3 \]

BB4:
\[ v4 = \phi\langle v1, BB1\rangle, \langle v2, BB2\rangle, \langle v3, BB3\rangle \]
\[ v4 = v5 \]

Problem: Introduce lots of copies for the coalescer
Design of the Register Allocator

- PHI Elimination
- Two-Address
- Coalescing
- Linear Scan Allocator
- Spiller
- Rewriter
Two-Address Pass

Convert SSA 3-address instructions into instructions with read-modify-write operands

\[ v1 = v2 \]
\[ v1 = \text{add } v2, v3 \]
Design of the Register Allocator

- PHI Elimination
- Two-Address
- Coalescing
- Linear Scan Allocator
- Spiller
- Rewriter
Register Coalescing

Eliminate copies by registers renaming

\[ v_1 = \ldots \]

\[ v_2 = v_1 \]

\[ \ldots = v_1<\text{kill}> \]

\[ \ldots = v_2<\text{kill}> \]

• Implementation is very aggressive:
  - Does value numbering to coalesce live ranges that “conflict”

Safe to rewrite v2 to v1 even though the intervals overlap because v1 is not changed.
Why coalesce aggressively?

• Coalescing expects allocator to split later if needed

• Don’t trust random decisions from input:
  – Copies coming in are from PHI elimination

• Can be useful places to split to reduce register pressure, but:
  – cannot be trusted, miss many important cases
  – coalescing happens before we know true register pressure
Design of the Register Allocator

- PHI Elimination
- Two-Address
- Coalescing
- Linear Scan Allocator
- Spiller
- Rewriter
Linear Scan Register Allocator

Single pass over list of variable live intervals ordered by starting points

Starting from v1, followed by v2, v3, v4, and then v5

v3 live interval ends here, r3 is now available

v5 conflicts with v1, v2, and v4, spilling is required
Linear Scan Register Allocator cont.

• Picking spill candidate based on def / use “density”
• Backtrack to the starting point of the spilled live interval

v1  v2  v3  v4  v5

drawn diagram:

v4 is then broken into two smaller intervals that do not conflict v5
Linear Scan Register Allocator cont.

- Spiller and allocator share responsibilities:
  - Linear scan decides: which live interval to spill
  - Spiller decides: how the interval is spilled

- Major problem:
  - Spill code insertion is deferred until all of allocation is done
  - Major bookkeeping nightmare
Design of the Register Allocator

PHI Elimination → Two-Address → Coalescing

Linear Scan Allocator

Rewriter

Spiller
Rewriter

- Rewrite virtual registers to allocated physical registers
- Insert spill and reload code
- Also perform some micro-optimizations

Allocations:
- v1 -> EBX
- v2 -> EAX
- v3 -> EAX
- v4 -> FI#4, ECX
- ...

v2 = addri v1, 17
v3 = mulrr v4, vr2
...

EAX = addri EBX, 17
ECX = load [FI#4]
EAX = mulrr ECX, EAX
...

...
LLVM Register Allocation

- Motivation
- Overview
- **Optimizations**
- Future Work
Coalescing: Instruction Commuting

bb:
// v2 is livein
v1 = op
...
\textbf{v2 = add v2, v2<kill>}
...
v2 = v2
...
= v2<kill>

v1 live interval    v2 live interval

“add” is commutable
Forward substitute
Coalescing: Sub-registers

• Eliminate pseudo instructions to “extract” part of a register

• Critical for targets such as X86 which has registers that are part of larger registers
  • e.g. AL, AH are sub-registers of AX; AX is a sub-register of EAX

```
bb:
... v1 = op
... v2 = extract_subreg v1<kill>, 2
... = v2<kill><subreg#,kill>

bb:
... EAX = op
... = EAX<kill><subreg#2,kill>
```
Spilling: Fold Spills and Reloads

BB:

v1

v1.1 = load [fi#1]

v1.2 = add32rr v1.1, 3

store v1.2, [fi#1]

v2 = add v1.3, v1.5

v3 = xor v1.4, v1.3

v4 = and v1.5, v1.4

v1.3 = load [fi#1]

v2 = add32rr v2, v1.3

v4 = and32rr v4, v1.5
Spilling: Splitting at BB Boundaries

v1.1 = load [fi#1]

v2 = add32rr v1.1, 3
store v1.2, [fi#1]

v1.3 = load [fi#1]
v3 = xor32rr v3, v1.4

v1.4 = load [fi#1]
v4 = and32rr v4, [fi#1]

v1.5 = load [fi#1]

High register pressure, spill again.
Machine LICM example visited: Simple Re-materialization

- Currently only re-materialize instructions with no register operands
- Hacked to allow PIC base register operands

```
loop_preheader:
v1 = VSET0  // xor xmm0,xmm0
store v1, [fi#1]
brcc bb1

bb1:
    ...
v1.1 = VSET[fi#1] xor xmm0,xmm0
    ... = v1.1
    ... brcc bb1
```

v1.1 = load [fi#1] = v1.1
Generalized Re-materialization

**loop_preheader:**

- v1 = load L_GV$stub
- v2 = load v1
- brcc bb1

**bb1:**

- v3 = add32rr v2, k
- \[= v2\]
- brcc bb1

**bb1:**

- v1.1 = load L_GV$stub
- v2.1 = load v1
- v3 = add32rr v2.1, k
- \[= v2.2\]
- v1.2 = load L_GV$stub
- v2.2 = load v1.2
- \[= v2.2\]
- brcc bb1

**bb1:**

- v1.1 = load L_GV$stub
- v2.1 = load v1
- v3 = add32rr v2.1, k
- \[= v2.2\]
- v1.2 = load L_GV$stub
- v2.2 = load v1.2
- \[= v2.2\]
- brcc bb1

- Need alias analysis information for load motion
- Must track available values / register

*Note v1.1 is available here!*
LLVM Register Allocation

• Motivation
• Overview
• Optimizations
• Future Work
Goals

- Faster compile times
- Generate faster code
- More maintainable and flexible code generator
Strong PHI Elimination

- Perform PHI elimination less naively
- Less work for the coalescer, compile-time benefit

\[
\begin{align*}
V_3 &= \ldots \\
V_1 &= \ldots \\
V_3 &= V_1 \\
V_2 &= V_3
\end{align*}
\]
Next Step: Iterative Splitting

- Split on loop boundaries
- Split on basic block boundaries
- Intra-block splitting

**loop_preheader:**

\[ v_1.1 = \text{load [fi#1]} \]

1. Insert reload in loop preheader

2. Insert reloads to start of BB’s

3. Insert reloads before a number of close uses.
Aggressive Re-materilization

loop_preheader:
\[ v1 = X + 4 \]
\[ v2 = X + 7 \]
\[ v3 = X + 15 \]

loop:
load [ iv + v1 ]
...
load [ iv + v2 ]
...
load [ iv + v3 ]

Re-materilize

Re-association
to enable sharing

Spilling due to register pressure

loop:
\[ v1.1 = X + 4 \]
load [ iv + v1.1 ]
...
\[ v2.1 = X + 7 \]
load [ iv + v2.1 ]
...
\[ v3.1 = X + 15 \]
load [ iv + v3.1 ]

loop:
\[ X.1 = iv + X \]
load [ X.1 + 4 ]
...
load [ X.1 + 7 ]
...
load [ X.1 + 15 ]
Backtracking in Linear Scan Allocator

• Two conflicting problems:
  − Assign registers aggressively to maximally use them, spilling when they run out
  − Spilling a use requires a register to reload into; a def must also target a register before it is spilled

• Problem: Backtracking is slow, requires redoing and undoing regalloc

• Solution: Add ability to spill previously allocated interval without backtracking, i.e. no more linear scan!
Summary: Much to be done!

• High Level Plans:
  – Smarter PHI elimination for faster compiles
  – Kill backtracking: Use iterative approach instead of linear scan
  – Maintainability: Insert spill code during spilling instead of after regalloc

• Improved Spilling:
  – Split live intervals at arbitrary places
  – Aggressive re-materialization in spiller
  – Use availability info to remat instructions with reg uses
  – Use alias Info to remat loads
  – Reschedule to reduce register pressure?

Questions?