Empty Book Template

Ryan Hou

Invalid Date

Table of contents

Preface

My notes on ???.

Resources

Some relevant resources:

- Resource Name

Textbooks:

- Book 1

1 Introduction

1.1 Perspective

> Note 1: Definition - Some definition
>
> **Term** is defined as blah blah blah...

This note does ...

1.2 High Level Ideas

2 Summary

Basic Block -

2.1 Control Flow Analysis

Control Flow Graph

Dominator

Immediate Dominator

Dominator Tree

Post Dominator

Immediate Post Dominator

Natural Loop

Back Edge

Loop Detection:

- Use dominator to find backedges
- Each backedge defines a loop
- Then merge all loops with same header

Parts of a Loop:

- Header
- Body (LoopBB)
- Backedges, exitedges, preheader (preloop), postheader (postloop)

Regions

Trace

Superblock

2.1.1 Predicated Execution

An alternative to branches: Predicated execution

- Hardware mechanism that allows operations to be conditionally executed
- Add an additional Boolean source operand called the predicate
- Only execute when the predicate is TRUE
- Benefits: No branches, can freely reorder independent instructions in the predicated block
- Costs: Need to execute all instructions, worst case schedule length, worst case resources required
- Guarding Predicate: Predicate that guards an instruction from being executed or not
  - Whether or not we're supposed to execute this instruction (1->execute, 0->fall-through)

If-conversion

Loop backedge coalescing

Control Dependence Analysis (CD)

Control Flow Substitution - going from branching code -> sequential predicated code

When to use If-Conversions?

Advantages

- Removes branches. Good if we have a 50/50 branch to get rid of mispredicts.
- Increase potential for operation overlap -> larger BBs so we can do more aggressive compiler transforms

Disadvantages

- High processor resource usage. Need to execute everything!
- Dependence height. Be careful with blocks with mismatched dependence heights
- Hazard presence for compilers, which can cause limited optimizations.

2.2 Dataflow Analysis

Live variable (liveness) analysis

Gen, Kill

Reaching definition analysis (rdefs)

DU/UD Chains

Transfer Function

Meet function

Available Definition Analysis (Adefs)

Available Expression Analysis (Aexprs)

2.3 SSA Form

Static Single Assignment Form

Phi Nodes

2.4 Code Optimizations

Dead Code Elimination Remove any operation who's result is never consumed Rules: - Cannot delete stores or branches - DU chain empty or destination register is not live This actually misses some dead code, especially in loops See algorithm in Lecture 8 for a better algorithm

Local Constant Propagation - Local means within a basic block - Maximally propagate constants/literals

Global Constant Propagation - Propagate across BBs

Constant Folding - Simplify operations whose source operands are all constants

Forward Copy Propagation - Forward propagation of the RHS of moves - Benefits: reduce chain of dependences and eliminate the move

Common Subexpression Elimination (CSE) - Eliminate re-computation of an expression by re-using the previous result - Make a new variable that stores the result of this subexpression. Replace uses of the subexpression with this new variable - Benefits: Reduce work/computation, moves can get copy propagated

Loop Invariant Code Motion (LICM) - Move operations whose source operands do not change within the loop to the loop pre-header - Execute them only once per invocation of the loop - Rules: - Only certain operations can be moved. No stores or branches - Source operand cannot be modified in the loop body - Instruction is the only instruction to modify the destination register - … more in Lecture 8 slides - If there is a load, need to make sure there are no writes to the same address in the loop

Global Variable Migration - Assign a global variable temporarily to a register for the duration of the loop - Load in preheader - Store at exit points - Global variables are mapped to memory,

but for the duration of the loop we may want to put it temporarily to a register so we're not constantly going to memory

Induction Variable Strength Reduction - Create basic induction variables from derived induction variables - Induction variable - Has a very specific pattern that we know ahead of time - BIV (i++) - basic induction variable - DIV (j = I * 4) - derived induction variable - Convert DIVs to BIVs. This reduces the chain of instructions and makes things simpler (potentially) - Issues: Initial and increment values, where to place increments?

Instruction-Level Parallelism (ILP) Optimizations - Instead of eliminating redundancy, this focuses on getting more parallelism and ability to overlap instructions - Increase ILP by breaking dependences

Back Substitution - Long expressions with many source operands can lead to the compiler frontend generating a long chain of sequential instructions - Apply Tree Height Reduction - Re-compute expression as a balanced binary tree

Loop Unrolling - Why unroll? -> We can get bigger loop bodies and allow for more opportunity for overlapping instructions -> more freedom to optimize and reorder - With known trip count (counted??), we can pick an unroll factor in a smart way, where the loop is unrolled a multiple of times to the unroll factor - For trip count that is not statically known (uncounted), we can peel off the first "odd" loop iterations. Create a preloop to ensure trip count of unrolled loop is a multiple of the unroll factor - Preloop computes how many non multiple loop counts we have - However, unrolling is not enough for overlapping different iterations together - We need register renaming between the unrolled iterations! - However, this is still not enough! - We need accumulator variable expansion § Create temporary accumulator variables § Each iteration targets a different accumulator § Sum up accumulator variables in the end § May not be safe for floating point operations! We can also do induction variable expansion

2.4.1 Acyclic Scheduling

2.4.2 Software Pipelining

2.4.3 Register Allocation

2.5

References