# Dynamic Compilation

do the dynamic compile once we know values of variables that will remain invariant thruout execution but unknown until then

more optimized than the more general case

# Dynamic Compilation

determines loop limit -->

- loop induction variable & any variables derived from it = RTC

whole point is performance

# Our Dynamic Compiler Framework

static compiler:

- generates machine code templates that assume they have been done; holes

dynamic compiler

- hardly a compiler, template manipulator

design analyses to support full functionality of C

# Dynamic Compilation System

see what opts we apply & effect of applying them

before how the framework applies them

capability today

# Cache Lookup Example

written as general program, able to handle

specific config at run-time: cache parameters = RTC

explain what it does

- internally

# Cache Lookup Example

red code calculates RTC & does opts based on RTC

our static analysis recognizes RTC; derive additional RTC from them: calculation whose operands ...

perform opts on them

variable reachable from a RTC = RTC

mult 2 RTCs = RTC; by constant folding obtain value

RTC controls loop termination; complete LU

offset is induction variable; different values in each iteration

# Cache Lookup Example

output of dynamic compiler framework: specialized for 4-way... (machine code)

recap what opts have been done

variable reachable from a RTC = RTC

mult 2 RTCs = RTC; by constant folding obtain value

determine if power of 2; yes $\Rightarrow$ replace expensive div by cheap right shift

replace mod by & (mask)

RTC controls loop termination; complete LU

offset is induction variable; different values in each iteration

result is half # instructions; half # loads

# Annotations

annotations

- for partially constant data structure, **dynamic** on memory dereference operators
- **key**: multiple compiled versions for a single DR, each optimized for diff RTCs
  hash lookup on value: pointer to DCed code or 0
- balance between manual (‘C) & automatic (no one)

termination condition is RTC

unrolling factor is reasonable or product

# Set-up Code

divide dynamic regions into run-time constant operations & control flow

vs. (partially) dynamic operations & control flow

constant code: set-up code

dynamic code: machine-code templates

holes: references to constant values

red code

calculates values of induction variable that are used as offset into array in each iteration

# Templates

executed each time dynamic regions is entered

can’t copy propagate RTCs outside the regions

# Dynamic Compiler Framework

table of RTC values; loops have a nested structure; follow the structure guided by directives

RA & CS for DR and surrounding code

- don't move holes outside template boundaries
- no copy prop of holes outside DR
- induction variables in unrolled loop are not loop invariant, so can't hoist: each iteration has own copy

back to the diagram

stitcher puts executable template into malloc'd space

- address of space in global var; indirect jump thru it
- another indirect jump out of it

# Dynamic Compiler Framework

Reachability analysis

- if dynamic branch, after merge x= RTC only if has **same** constant reaching def on both paths
- if static branch, only one path will be taken; want x to be RTC
- determines restricted conditions (expressed as branch outcomes) under which program points can be reached after dynamic compilation
- at merge: reachability analysis determines if conditions are mutually exclusive (union)
- loop head merges have mutually exclusive conditions, each iteration has its own code

use both sets of RTCs & reachability conditions for previous program point to calculate for current program point

# Preliminary Results

different values of x & y

800x800 multiplied by all scalars 1..100: the average

# Preliminary Results

eager: dynamic branches, structured control flow

lazy set-up & stitching: defer until path is executed the first time (unstructured control flow or code blowup if eager: multiway LU)

6-10 instructions/instruction generated for structured code with peephole opts

programs with RTC offsets to array elements: SW stack in calculator

directives for removing & modifying instructions if array element is stored in reg

- eliminates ld/st/address arith
- 1.7 $\Rightarrow$ 4.1
- new annotations: hints for register actions
- also floats & pointers

rematerialization: saves loading from table

SWP: template of a few unrolled loops

# Future Directions

constant within a region; changes between region invocations

- caching vs redynamically compiling vs general code
- analysis for quasi-invariants would also ensure correctness with 1 RTC value

# More Future Directions

optimization decision-making to static compiler; patching & copying

with higher level info to support desired run-time optimizations

- CS across template boundaries, especially unrolled loops

based on types of opts that can be applied & expected frequency of execution

# More Future Directions

background:

- speedups varied across programs, across platforms: each machine best & worst for different programs
- architectures similar
- load elimination depends on # cache ports & makeup of multiple instruction issue
- cheap instructions eliminated, writes left: size of write buffer a factor

feasibility studies:

- isolate key components & experiment with alternative designs
- what types of instructions are eliminated?
- what effect on multiple instruction issue?
- what effect on cache performance (code placement in L1)?
- what effect on write buffer performance?
- what effect on branch performance (eliminates unpredictable branches)?

# More Future Directions

- alias & side-effect analyses

# Summary

completes machine-code template construction

# Related Work

## PE

- phased compilation
- BTA + static analyses, specialization done in stitcher
- source-to-source, analysis on higher level ASTs
- purely functional languages

# Related Work

**Fabius** (Lee, Leone: CMU)

- partial evaluation framework
- first-order, purerly functional subset of ML
  no side effects (RTCs don't change)
- currying triggers DC: for each argument, new function is compiled
  BTA identifies calculations that depend on first argument
- safe because side effects disallowed
  our correctness depends on correct programmer annotations

# Related Work

**Consel & Noel:** Rennes

- partial evaluation framework
  produces C templates
- subset of C
  pointer analysis unsafe with casts & pointer arithmetic
- interprocedural
- analysis on syntax trees
  no unstructured control flow
  no reachability analysis

# Related Work

**'C** (Engler, Hsieh, Kaashoek: MIT)

- extension to C for specifying run-time constants & dynamic regions
- uses DCG: library for constructing expression trees