Comparison with Related Work

Partial evaluators

Leone & Lee [94]

Consel & Noël [96]

Engler, Hsieh, & Kaashoek [96]

Guenter, Knoblock, & Ruf [95]

Partial Evaluators

Partial evaluators are a general source-to-source transformation system

- find procedures of the form `proc(static, dynamic)`,
  where value of **`static`** is known at PE-time
- produce new version $\texttt{proc}_{\textbf{static}}\texttt{(dynamic)}$
- perform constant propagation, constant folding, branch elimination,
  loop unrolling, callee procedure specialization, and more,
  based on values derived from **`static`**

We fit into this framework

- static = run-time constant
- perform analysis statically, perform specialization at run-time
- optimizations performed by PE similar to ours

We have some differences:

- integrated into an existing optimizing compiler framework
- handle general imperative languages

Other PE-Based Dynamic Compilers (#1)

	Leone & Lee's Fabius	**Us**
input language	first-order, purely functional language with currying	Full C
annotations	currying implies dynamic compilation	dynamicRegion, unrolled
model of generated code	individual instructions	optimized machine-code templates
generated code quality	low, medium if run-time opts	high
dynamic compilation overhead	low, medium if run-time opts	fairly low

Other PE-Based Dynamic Compilers (#2)

	Consel & Noël	**Us**
input language	Full C; cannot handle unstructured constructs? unsafe in face of pointer arithmetic	Full C
annotations	like PE; does interprocedural analysis	dynamicRegion, unrolled
model of generated code	optimized machine-code templates	optimized machine-code templates
generated code quality	fairly high	high
dynamic compilation overhead	low	fairly low

Other Dynamic Compilers

	Engler *et al.*'s 'C	**Us**
input language	Full C	Full C
annotations	programmer explicitly manipulates code fragments imperatively & flexibly	annotations --> static compiler analysis
model of generated code	intermediate code trees	optimized machine-code templates
generated code quality	up to programmer; low to very high	high
dynamic compilation overhead	high	fairly low

Other Dynamic Specializers

Guenter *et al.* speed up an interactive graphics rendering application

- user varies one parameter, keeps many others fixed

Statically produce a small number of specialized shading routines,
one pair per parameter

- one routine computes & caches intermediate values derived solely from fixed parameters
- a second routine combines the varying parameter with the cached values

Dramatic speed-ups (> 100x) possible

Could run even faster if using dynamic compilation

Future Directions

Explore and expand applications suite

- numerical & scientific code
- systems code
- parallel codes
- ...

Extend basic framework

- handle run-time constants that can change
- handle interprocedural dynamic regions
    - need a better compiler infrastructure!
- support inlining
- support unrolling loops over tree, dag, and graph data structures
- more, as motivated by applications

More Future Directions

Investigate alternative divisions of labor between static & dynamic compilers

Basic trade-off: dynamic compiler can do more optimization

$\Rightarrow$ better quality code

$\Rightarrow$ slower dynamic compilation

Can enrich run-time representation of programs
to support desired run-time optimizations

Ideas:

- static compiler generates specialized intermediate representation designed to support selected dynamic optimizations quickly
- for different regions, make different trade-offs,
based on benefits & costs of dynamic optimizations for each region

More Future Directions

Explore automating dynamic compilation

Key ingredients:

- dynamic profile information
- static analysis of potential benefits of dynamic compilation
- alias & side-effect analyses