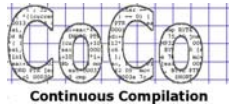


Continuous Compilation for Aggressive and Adaptive Code Transformation



Bruce R. Childers

University of Pittsburgh
Pittsburgh, Pennsylvania, USA
childers@cs.pitt.edu
<http://www.cs.pitt.edu/coco>

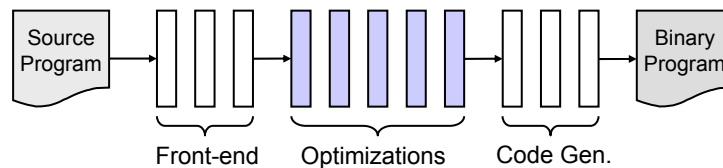


Today's talk is based on:
Min Zhao's Ph.D. thesis, co-advised with Mary Lou Soffa



Sponsored by Next Generation Software, National Science Foundation

Code Optimization



- **Critical role:** Translate high-level language programs into an efficient form to run on target machine
- Sophisticated optimization algorithms exist and do quite well – over 40 years of experience!
- Yet, we don't really understand **how to use** optimizations – in some sense, we are lucky

Using Optimizations

- Using optimizations
 - Degrade performance in some situations – *where to apply?*
 - Impact by disabling and enabling opportunities – *what order?*
 - Parameters lead to wide variance – *how to configure?*
- Why now?
 - Embedded systems use high-level languages
 - Dynamic optimization useful, but cost is very high
 - Performance improvements are shrinking
- **Our Goal:** *Look for new opportunities and develop more effective ways to apply optimizations*



Continuous Compilation

- Apply optimizations both statically at compile-time and dynamically at run-time with optimization planning at compile-time
- Plan for both static and dynamic optimizations
 - Understand properties of existing optimizations
 - Efficacy of both static and dynamic optimizations
- **Determine where to apply optimizations, which ones to apply, the order in which to apply them, and their parameters**

Continuous Compilation

Prediction & Planning

Static Compilation (using profiles, estimation models)

Phase 1

Dynamic Optimization

Program Execution and Dynamic Optimization (using monitor and optimization plans)

Phase 2

Off-line Adaptation and Refinement of Monitor and Optimization Plans

Phase 3

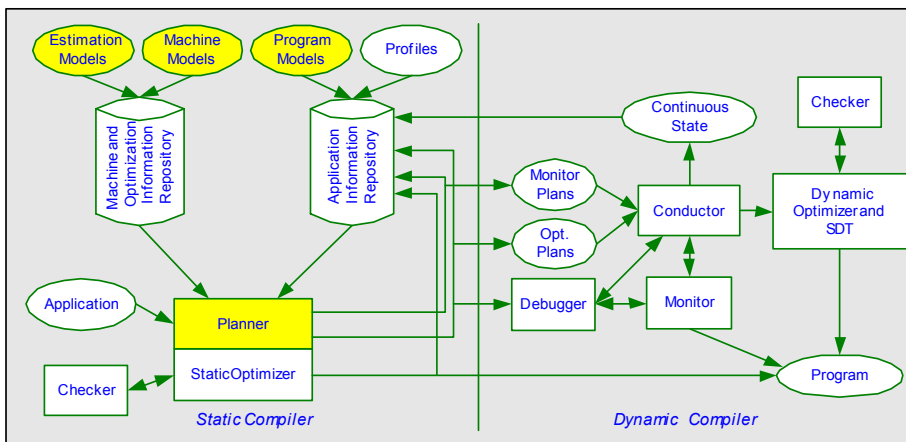
Recompilation and Regeneration of Monitor and Optimization Plans

Phase 4

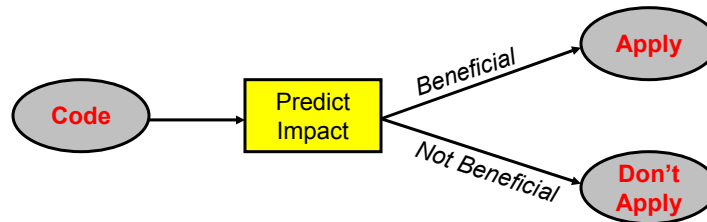
As time passes, the continuous compiler moves through phases, possibly revisiting earlier ones.

Target applications: Long running programs that have different phases of execution.

CoCo Architecture



Prediction of Profit and Cost



- **Predict the impact and costs of optimization**
- With estimates of benefit and penalty, answer the questions:
 - Where and what optimizations to apply?
 - In what order to apply?
 - What optimization parameters?

Challenges

- Performance varies widely, based on
 - Code context (e.g., loop trip count)
 - Parameters of optimizations (e.g. loop unrolling factor)
 - Machine configurations (e.g. cache configuration)
 - The order of optimizations
- Many resources impact overall performance
 - Cache configuration
 - Instruction scheduling rules
 - Register numbers and types
- Thus, depends on: **Code, optimizations, & machine**

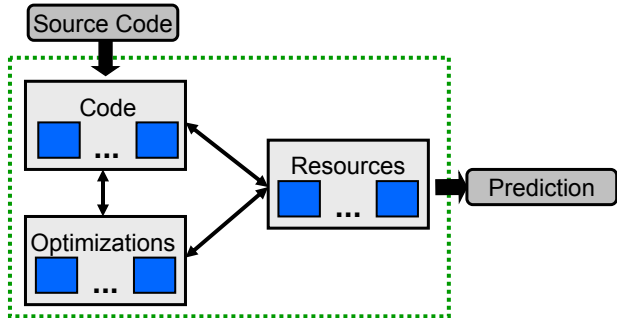
Traditional Approach

- Usually, applied in an *ad hoc* fashion
 - Heuristics to guide when to apply
 - E.g., Always apply if applicable
 - E.g., Apply if enough registers available
 - Predetermined order to apply optimizations
 - Certain orders just “make sense”
 - Compiler writer’s experience
 - Fixed configurations [Triantafyllis03]
 - Analytic/experimental for loop opt. [Chen05, Yotov03, Ding03, Wolf91, Sarkar97, McKinley96, Hu02, Zhao03, ...]
- ⇒ **Missed opportunities, bad decisions, limited to certain optimizations, tuning, etc.**

Our Approach

- Build and develop analytic models to **predict** profit/penalty, **without** actually applying the optimization
 - Need models of particular optimizations
 - Need models of the code
 - Need models of the resources that are effected
- Based on models, make decisions about how to apply optimizations
 - We don’t need accurate models, just the trend needs to be accurate enough to do the estimates

FPO: Framework for Predicting Optimizations



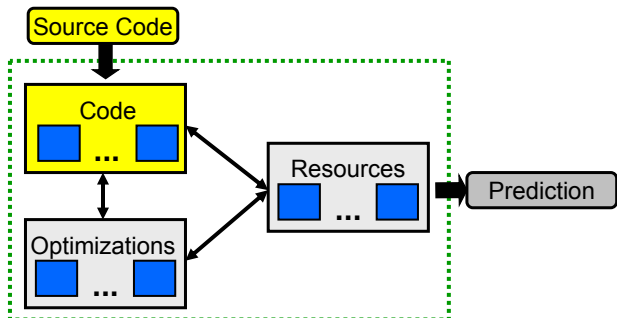
Plug 'n Play Models

1. Extract code context
2. Model effect of the optimization
3. Model effect on the machine resources

FPO: a **F**ramework, consisting of models, for **P**redicting the impact of **O**ptimizations

Models for both *scalar opts* [CGO'05] and *loop opts* [LCTES'03]

Code Model



Example Characteristics

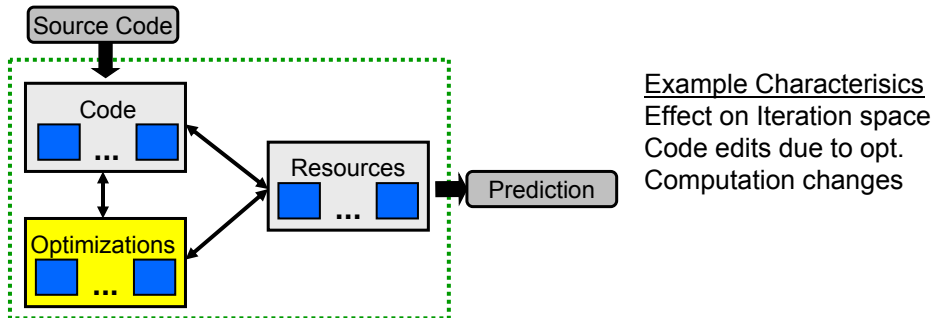
Iteration space
References
Register pressure
Computation

Express **code characteristics** relevant to opt. and resources

Automatically constructed from program

Abstracts only relevant details – it is not an intermediate rep.

Optimization Model

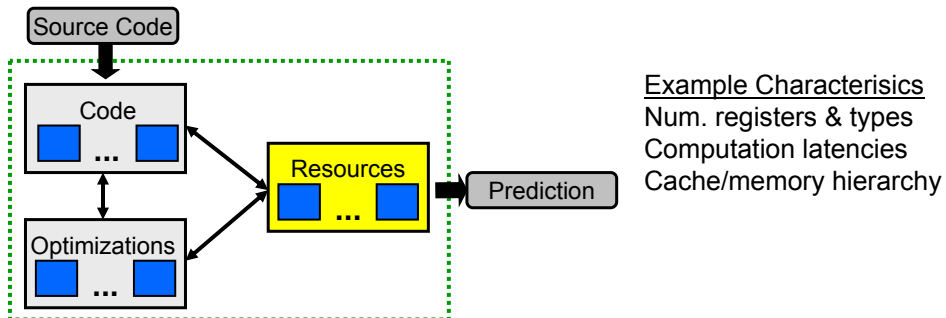


Example Characteristics
Effect on Iteration space
Code edits due to opt.
Computation changes

Model **how optimization transforms code model**

Describes optimization semantics – how the code model is affected by the predicted application of the optimization

Resource Model

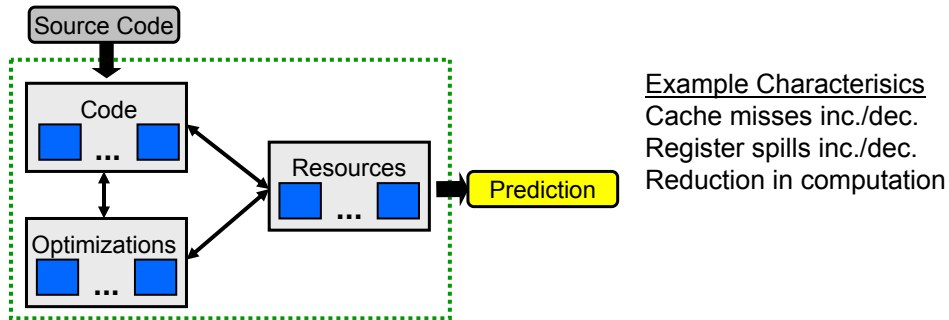


Example Characteristics
Num. registers & types
Computation latencies
Cache/memory hierarchy

How the **code model effects machine resources**

What machine resources are available & abstract only details impacted by the code (and optimization) model

Predicted Optimization Profit



Example Characteristics
Cache misses inc./dec.
Register spills inc./dec.
Reduction in computation

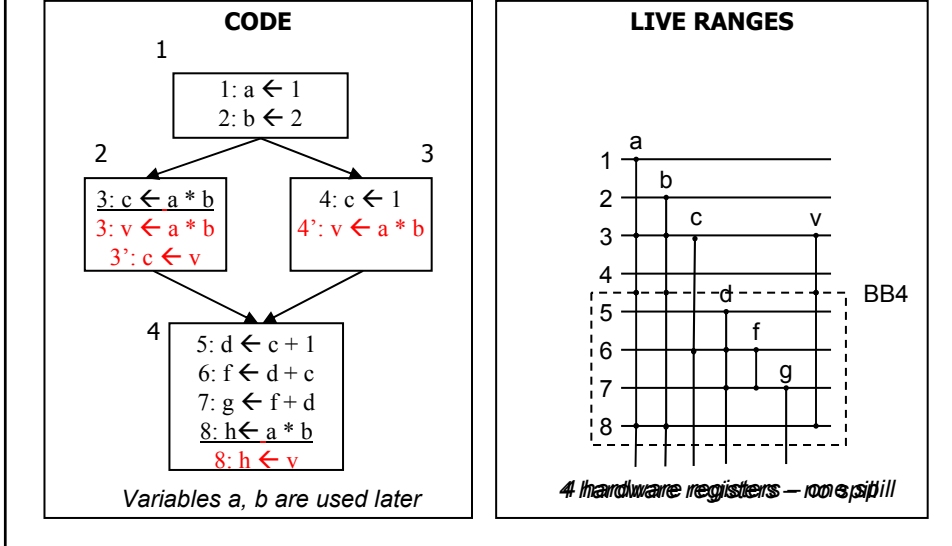
Compare transformed & non-transformed code

Prediction of the **profit** (or penalty) is the difference between unoptimized and optimized

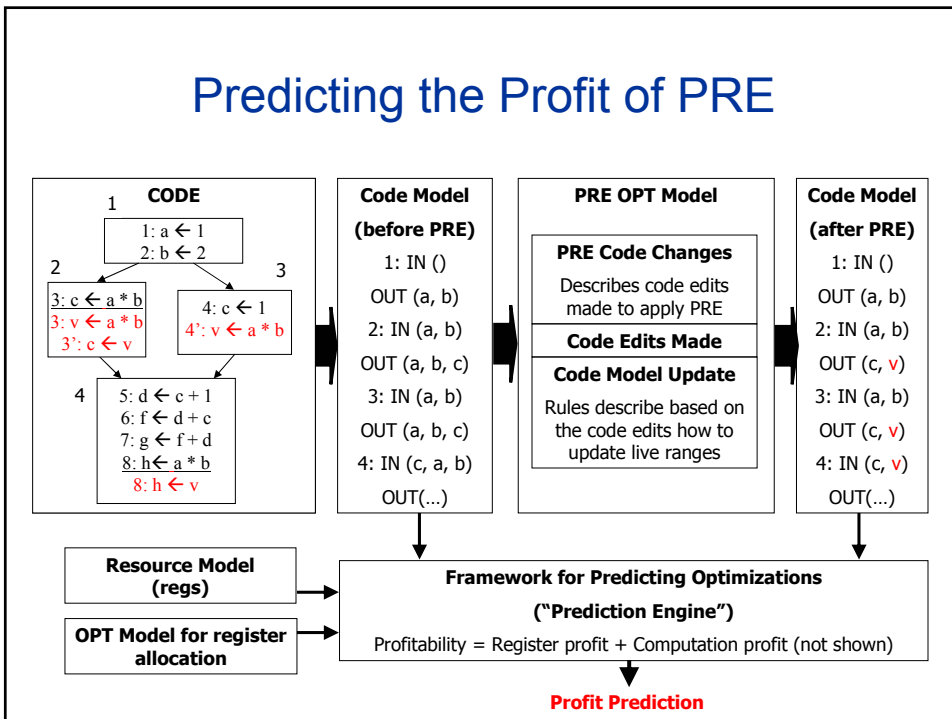
FPO: Scalar Optimizations

- Transformations that operate on scalar code
 - E.g., constant propagation, dead code elimination, **partial redundancy elimination**
- Can have several impacts
 - Reduce amount of computation
 - Change register pressure (for the better *or* for the worse!)
 - May change memory referencing pattern and cache behavior
- FPO (initially) considers
 - Affect on computation
 - How register pressure helps or hurts spills and reloads

Partial Redundancy Elimination



Predicting the Profit of PRE



PRE Optimization Model

- Describes semantic affects as code edits

IF partial redundant expression exp :

$T = exp$ is partially redundant at [Block Bs , Statement Ss]

move exp to [Bd , Sd] & assign temporary V ;

T_1 at [Ba_1, Sa_1] ... T_n at [Ba_n, Sa_n] are redundant

Moves expression to a new location

THEN

step 1: <Ins exp USE [Bd, Sd]>, <Ins V DEF [Bd, Sd]>

step 2: <Del exp USE [Bs, Ss]>, <Ins V USE [Bs, Ss]>

Delete expression; insert copy (V) at this location

step 3: forall T_i at [Ba_i, Sa_i]:

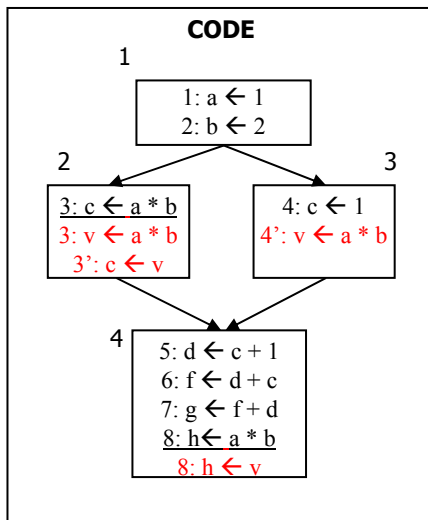
<Del T_i DEF [Ba_i, Sa_i]>, <Ins V DEF [Ba_i, Sa_i]>

<Ins T_i DEF [Ba_{i+1}, Sa_{i+1}]>, <Ins V USE [Ba_{i+1}, Sa_{i+1}]>

Replace redundant destination (D) with temporary V & insert a copy to original destination (e.g., $D=V$)

Opt. edits [Whitfield97]

PRE Opt. Model Determines Edits



EDITS FROM PRE OPT MODEL

Block 3 (from Step 1)

Ins a USE, Ins b USE, Ins v DEF

Block 4 (from Step 2)

Del h DEF, Del a USE, Del b USE

Ins v USE, Ins h DEF

Block 2 (from Step 3)

Del c DEF, Ins v DEF

Ins v USE, Ins c DEF

Incremental Code Model Update

- For registers, edits affect the live ranges
- Rules determine how to update live ranges

Code Change

Insert a use u of variable v in block B at statement s

Code Model Update

if v is live at $post-s$: no change;
 else if there is a use or definition before s in block B :
 no change to global code model; record local change
 else
 add v to $IN(B)$ and all reachable predecessors of B ;

A code edit on a variable use or def

How to update the live ranges

Incremental data flow [Pollock89]

Edits Determine Code Model Update

EDITS FROM PRE OPT MODEL

Block 3

Ins a USE, Ins b USE, **Ins v DEF**

Block 4

Del h DEF, **Del a USE, Del b USE**

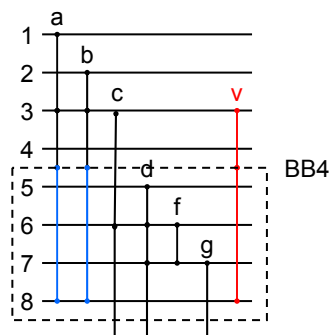
Ins v USE, Ins h DEF

Block 2

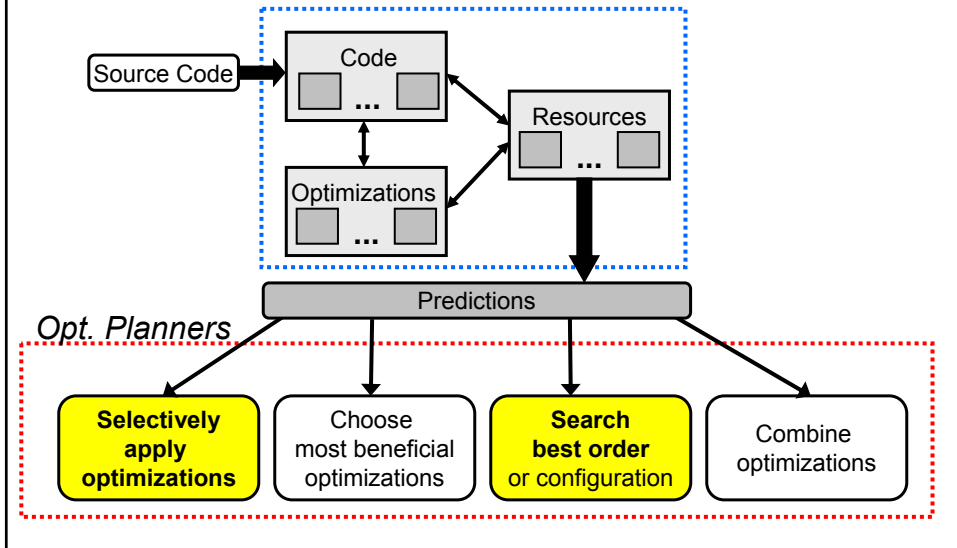
Del c DEF, **Ins v DEF**

Ins v USE, Ins c DEF

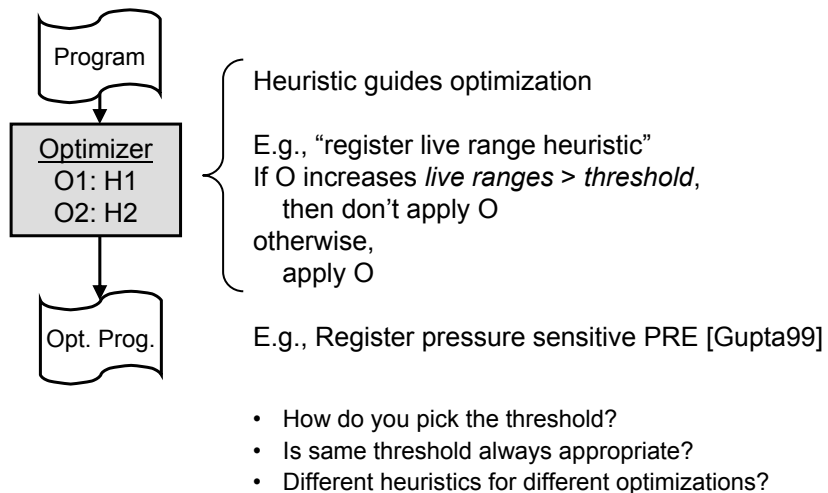
CODE MODEL -- LIVE RANGES



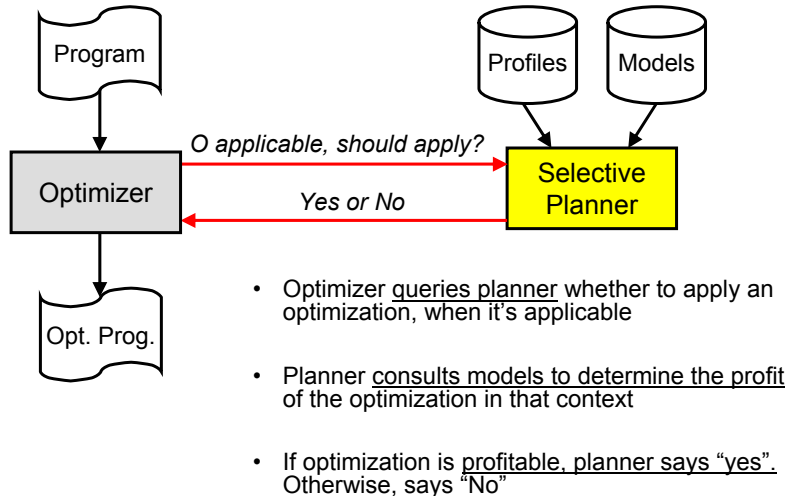
Using FPO for Optimization Planning



Traditional Approach: Selectively Applying Optimizations



FPO Approach: Selectively Applying Optimizations



Experimental Results

Experiments

- Model accuracy
- Heuristic approach when applying a single opt
- Selective planner when applying a single opt

Setup

- Benchmarks: SPEC2K, MiBench, MediaBench
- Platform
 - MachSUIF compiler, implemented models & optimizations
 - x86 AMD Athlon 1.4 GHz, 2 GB memory, RedHat Linux
- PRE and LICM (have done others)

PRE & LICM Model Accuracy


Benchmark	PRE			LICM		
	TP	CP	%Accacy	TP	CP	%Accacy
<i>gzip</i>	48	43	89.58	45	38	84.44
<i>vpr</i>	303	291	96.04	230	217	94.35
<i>mcf</i>	51	44	86.27	52	43	82.69
<i>parser</i>	293	210	87.87	75	68	90.67
<i>vortex</i>	530	431	81.13	346	303	87.57
<i>bzip2</i>	56	44	78.57	88	79	89.77
<i>twolf</i>	475	433	91.12	345	306	88.70

Correct predictions: Did model predict same as actual execution runs

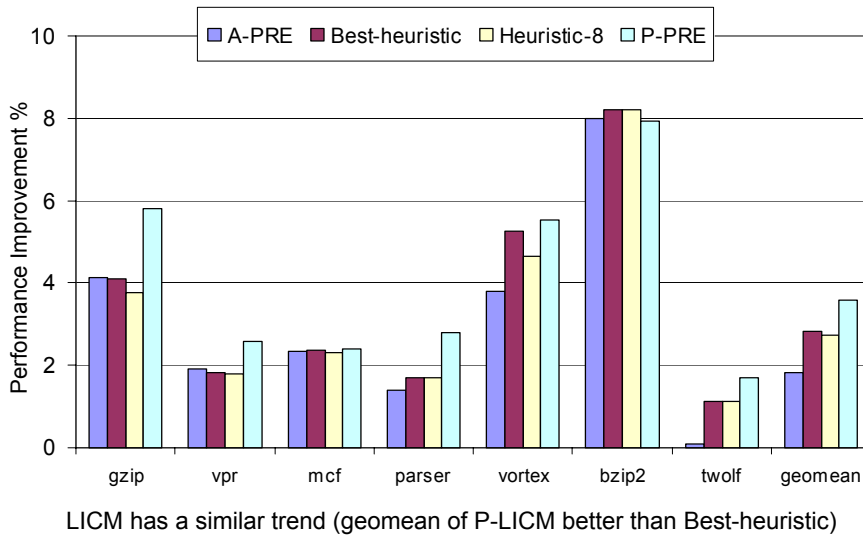
PRE & LICM with Heuristic

Benchmark	Heuristic-driven PRE				Heuristic-driven LICM			
	0	4	8	16	0	4	8	16
<i>gzip</i>	3.50	3.75	3.78	4.10	2.90	3.29	5.40	3.27
<i>vpr</i>	1.22	0.75	1.81	1.83	-0.40	-0.38	0.52	0.69
<i>mcf</i>	2.37	2.35	2.31	2.22	2.50	2.62	2.58	2.47
<i>parser</i>	1.25	1.50	1.70	1.35	2.55	2.86	1.99	2.23
<i>vortex</i>	4.73	5.25	4.66	3.86	4.88	5.69	4.99	5.28
<i>bzip2</i>	7.35	7.52	8.19	7.91	7.02	7.35	6.70	4.57
<i>twolf</i>	1.07	0.88	1.14	0.02	0.52	0.38	2.14	1.91

Results are percentage improvement

 Best performance

PRE Heuristic vs. Selective Planner



Planning for Optimization Order

- Can treat as a search problem: Many orders possible, but which is the best?

Adaptive optimization: Change order to fit context

1) Iterative compilation [Knijnenburg03]

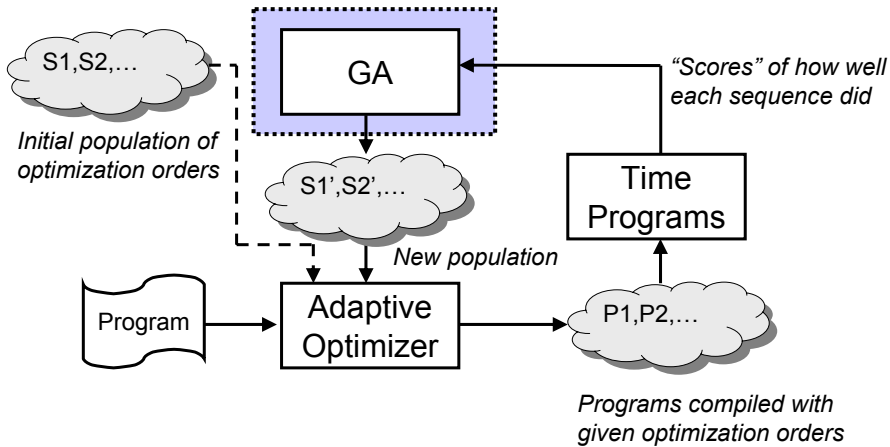
- Try an order, run it, feedback, try again

2) **Genetic algorithm** [Cooper99], [Almagor04], [Kulkarni04]

- Try many orders, select fittest, try again

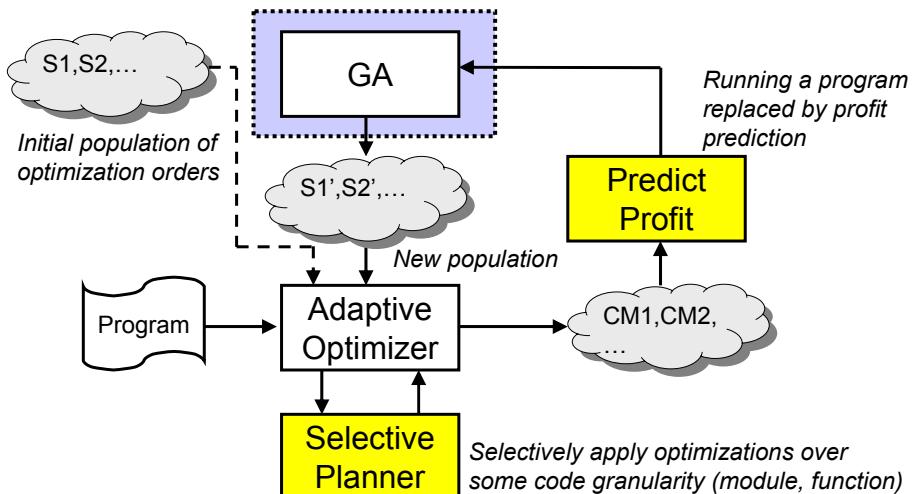
- Many granularities: Whole program, function, code region

Adaptive Optimization



Cooper [LCTES 1999], Almagor [LCTES 2004], Kulkarni [PLDI 2004]

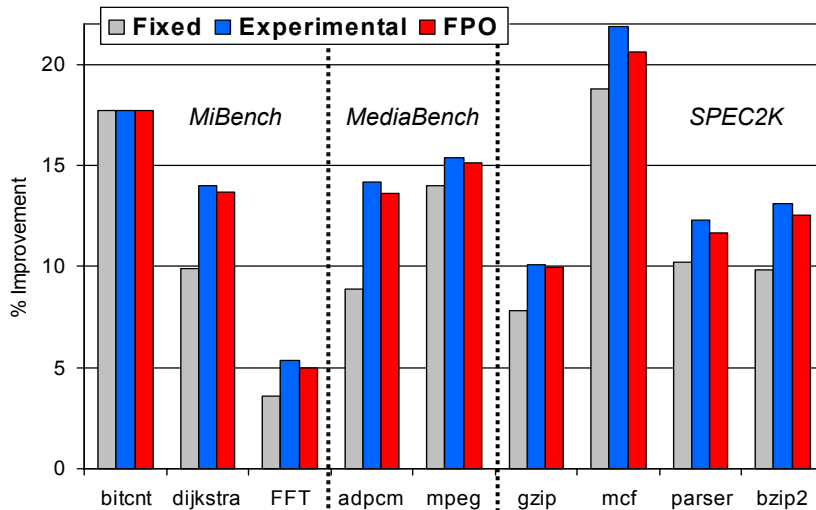
Using FPO for Adaptive Optimization



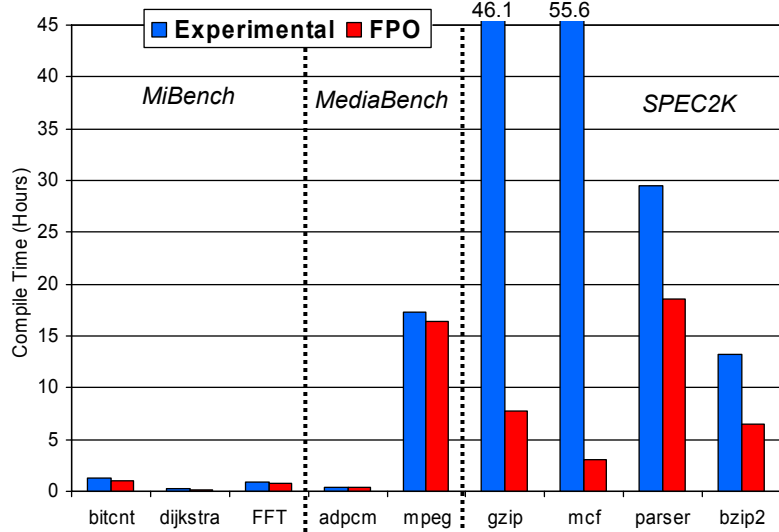
Experimental Results

- Scalar optimizations
 - Global value numbering (G), Constant propagation (C), Copy propagation (O), Constant folding (N), PRE (P), LICM (L), Register allocation (graph coloring) (R), dead code elimination (D)
- Fixed sequence: GOCDPOLOD [Whitfield97]
- GA based on [Almagor04]
- 10 generations, 20 sequences per generation
 - Initial sequence is fixed order sequence
 - 10% best sequences survive each generation, remaining formed by crossover operation and character-by-character with 5% mutation rate
 - Past sequence results are hashed to avoid re-evaluating
 - Optimization orders determined on module level

Run-Time Performance

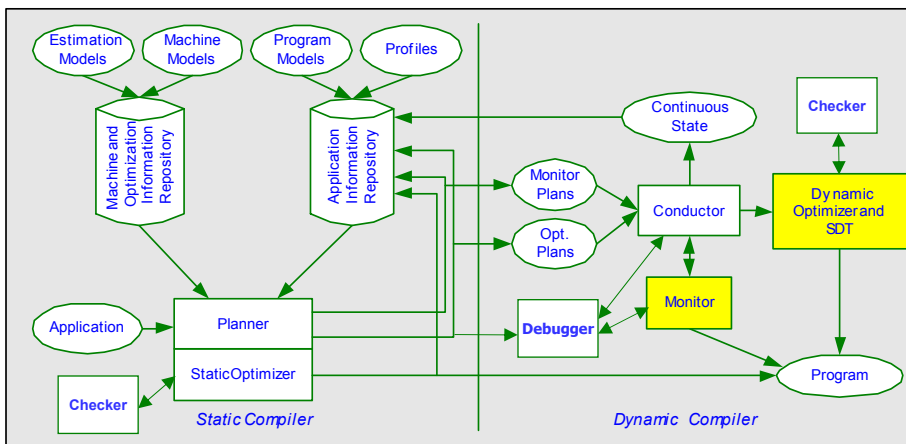


Compile-Time

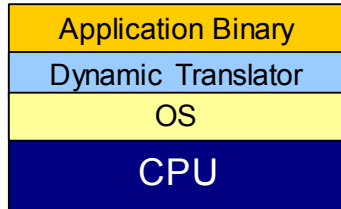


Slide 40

CoCo Architecture

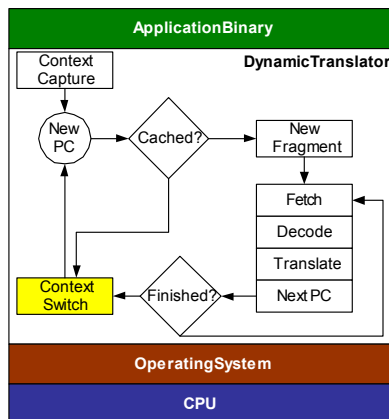


CoCo Run-Time System



- Based on *Software Dynamic Translation*
- Layer of software between application binary and the OS/CPU.
- Application's instructions are examined and modified before being executed on the CPU.
- Uses include binary translation, dynamic optimization, & others

CoCo Run-Time System



- **Strata toolkit** [CGO'03,'05 Tutorial]
 - Application's instructions are examined and modified before being executed.
 - Reconfigurable & retargetable
 - Low overhead translation
- Provides functionality for run-time translation & optimization
 - Multithreading & interrupt handling
 - Memory management
 - Translated code caching
 - Code analysis

Targets: MIPS/Irix, SPARC/Solaris, x86/Linux, MIPS/PS2, PPC/MacOS
 Apps: Code compression [DATE'04], self managing systems [WOSS'04], distributed execution [VEE'05], architecture simulation, program profiling

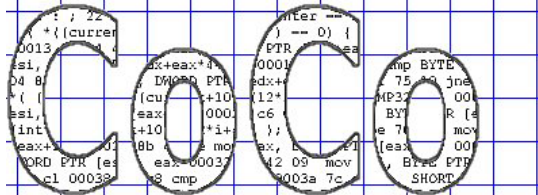
Strata SDT Toolkit

- Efficient translated code [CGO'03,IJPP]
 - Fragment linking, partial call inlining, fast returns, indirect branch translation caching, instruction trace formation
 - SPARC: 22.9x (no transformations) to 1.3x (with transformations)
 - Strata-x86: 1.3x vs. Dynamo/RIO [Bruening'03] 1.2x
- Program instrumentation [ICSE'05,WOSS'04,Traces'03]
 - Reduce amount of instrumentation inserted
 - Reduce the cost of an individual piece of instrumentation code
 - 1.26x-2.63x speedup over no instr. optimization for several profilers
 - ATOM [Srivastava'94], FIT [De Bus'04], Pin [Tutorial ASPLOS'04]
- Source-level debugging of dynamically translated code
 - Algorithms to track dynamic translations & map to source code
 - Integrated GDB and Strata – very efficient: 1.4x to 2.1x & < 1.6 MB

Summary

- Planning-based approach to compilation
 - Understanding profitability property
 - Models **do** work! Can successfully guide optimizer
 - Making adaptive optimization feasible
- Dynamic translation
 - Strata dynamic translator [CGO'03, IJPP & CGO'05 tutorial]
 - Dynamic instrumentation & monitoring [ICSE'05, WOSS'04]
 - Dynamic instrumentation optimization
 - Debugging dynamically translated programs

More Information



Continuous Compilation

childers@cs.pitt.edu

<http://www.cs.pitt.edu/coco>