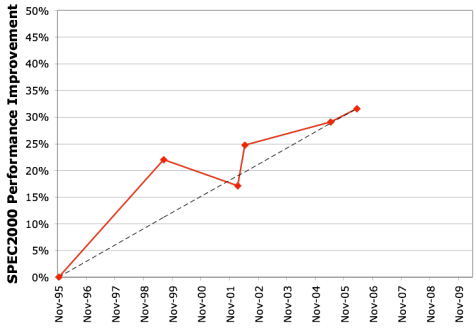
Towards a More Principled Compiler: Progressive Backend Compiler Optimization

David Koes

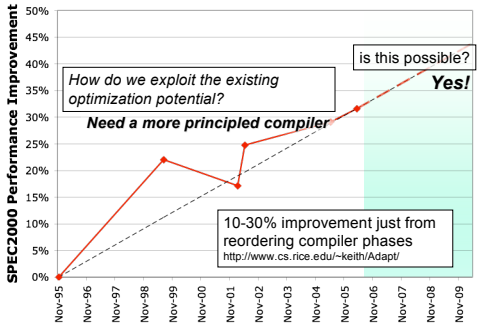
8/28/2006

Performance Gains Due to Compiler (gcc)

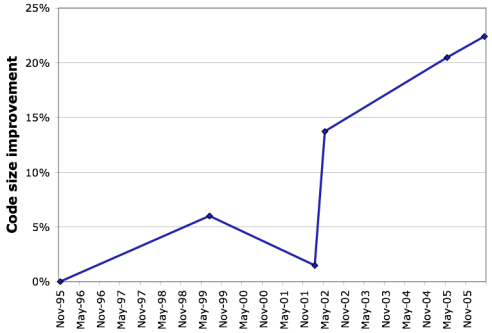


2.8Ghz Pentium 4, 1GB RAM, -O3 ...

The Future of Compiler Optimization

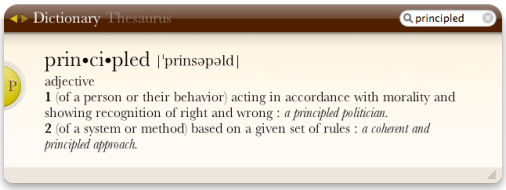


Compiler code size improvement



Carnegie Mellon School of Computer Science

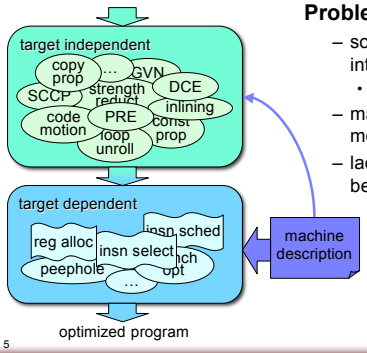
A Principled Compiler



A compiler that
- knows right from wrong
  (less optimal from more optimal)
- follows a rigorous procedure to get the desired output

Carnegie Mellon School of Computer Science

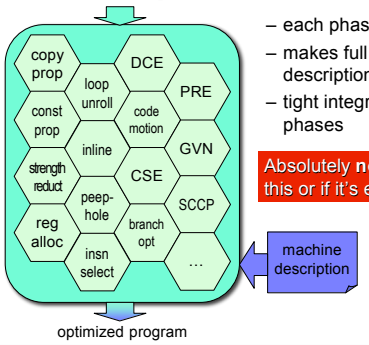
Today's Compiler



Problems
- some phases not internally optimal
  - purely heuristic solution
- machine description mostly ignored
- lack of integration between phases

Carnegie Mellon School of Computer Science

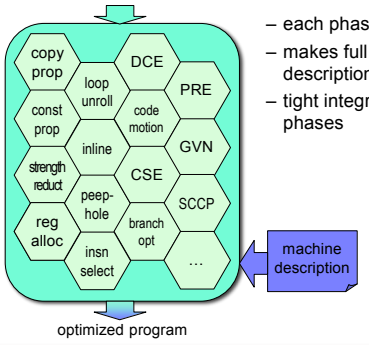
Ideal Compiler



- each phase locally optimal
- makes full use of machine description
- tight integration between phases

Absolutely **no** idea how to do this or if it's even *possible*

Towards a More Principled Compiler



- each phase locally optimal
- makes full use of machine description
- tight integration between phases

Outline

I. Motivation
II. Related Work
III. Completed Work
IV. Proposed Work
V. Contributions & Timeline

Register Allocation Problem

unbounded number of

limited number of processor registers + slow memory

spill code optimization

```
v = 1
```

rematerialization

```
u = v
t = u + x
```

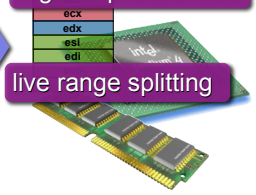
memory operands

```
print(t);
print(u);
...
```

register allocator

register preferences

live range splitting



Carnegie Mellon School of Computer Science

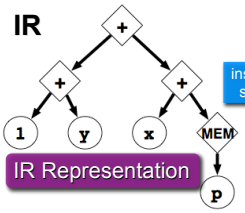
Register Allocation Previous Work

Method	Expressive	Fast	Optimal
Linear Scan	✘	✔✔	✘
Graph Coloring	✘	✔	✘
Integer Linear Programming	✔	✘	✔
Partitioned Boolean Quadratic Programming	✔	✔/✘	✘/✔

Carnegie Mellon School of Computer Science

Instruction Selection Problem

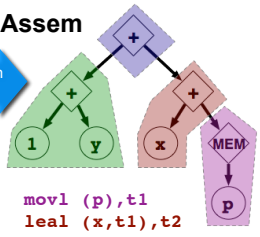
IR



IR Representation

instruction selector

Assem




```
movl (p),t1
leal (x,t1),t2
leal 1(y),t3
leal (t2,t3),r
```

minimum cost tiling

?

Carnegie Mellon School of Computer Science

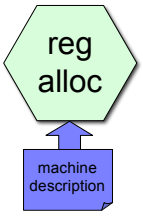
Instruction Selection Previous Work

Method	DAG Tiling	Register Allocation Aware	Fast	Optimal
Dynamic Programming	✘	✘	✔	✔
Binate Covering	✔	✘	✘	✔
Peephole Based Instruction Selection	✔	✘	✔	✘
AVIV Code Generator	✔	✔	✘	✘
Exhaustive Search	✘	✔	✘	✔

Outline

I. Motivation
II. Related Work
III. Completed Work
IV. Proposed Work
V. Contributions & Timeline

A More Principled Register Allocator



– fully utilize machine description
  - ***explicit* and *expressive* model of costs of allocation for given architecture**
– optimal solutions

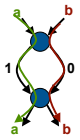
Multi-commodity Network Flow: An Expressive Model

Given network (directed graph) with
- cost and capacity on each edge
- sources & sinks for multiple commodities

Find lowest cost flow of commodities

NP-complete for integer flows

Example:
edges have unit capacity



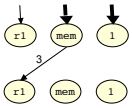
Register Allocation as a MCNF

Variables → Commodities

Variable Definition → Source 

Variable Last Use → Sink 

Nodes → Allocation Classes (Reg/Mem/Const)

Registers Limits → Node Capacities 

Spill Costs → Edge Costs

Allocation → Flow

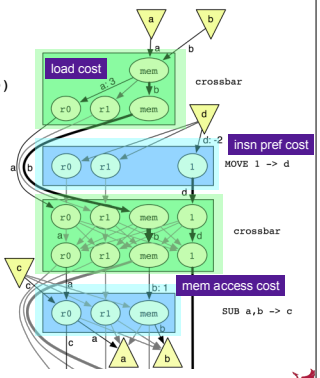
Example

Source Code

```
int example(int a, int b)
{
   int d = 1;
   int c = a - b;
   return c+d;
}
```

Pre-alloc Assembly

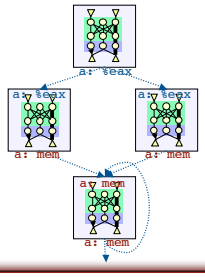
```
MOVE 1 -> d
SUB a,b -> c
ADD c,d -> c
MOVE c -> r0
```



COMPLETED WORK

Control Flow

MCNF can only represent straight-line code
– need to link together networks from basic blocks



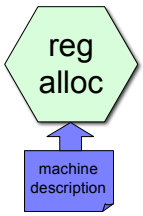
Extend MCNF model with merge *and* split *nodes to implement boundary constraints.*

details in proposal document…

along with modeling persistence of values in memory

COMPLETED WORK

A Better Register Allocator

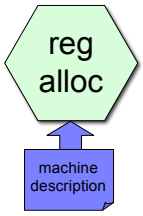


– fully utilize machine description
- ***explicit* and *expressive* model of costs of allocation for given architecture: Global MCNF**

– locally optimal
- NP-hard, so use *progressive* solution technique

COMPLETED WORK

A Better Register Allocator



– fully utilize machine description
- *explicit* and *expressive* model of costs of allocation for given architecture: **Global MCNF**

– locally optimal
- **NP-hard, so use *progressive* solution technique**

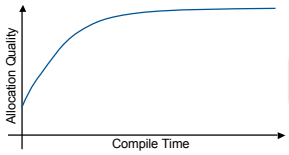
COMPLETED WORK

Progressive Solution Technique

Quickly find a good allocation

Then progressively find better allocations

- until optimal allocation found
- or time limit is reached



Technique:

Lagrangian relaxation directed allocators

COMPLETED WORK

Lagrangian Relaxation: Intuition

Relaxes the hard constraints

- only have to solve single commodity flow

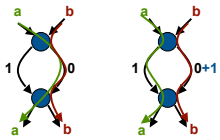
Combines easy subproblems using a Lagrangian multiplier (price)

- an additional price on each edge
- a price on each split/merge node

Example:

edges have unit capacity

with **price**, solution to single commodity flow *can be* solution to multicommodity flow

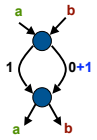


COMPLETED WORK

Solution Procedure

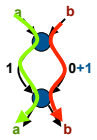
Compute prices with iterative subgradient optimization

- guaranteed converge to optimal prices
- optimal for linear relaxation



At each iteration, construct a feasible integer solution using current prices

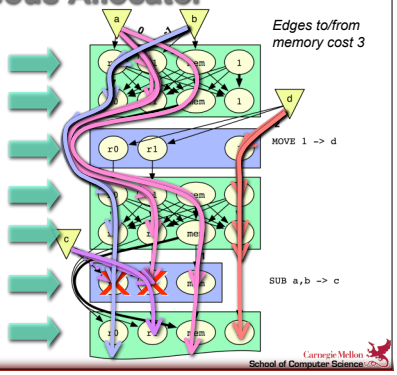
- iterative allocator *in document*
- simultaneous allocator
- trace-based simultaneous allocator



COMPLETED WORK

Simultaneous Allocator

Current cost:
–2



24

Carnegie Mellon School of Computer Science

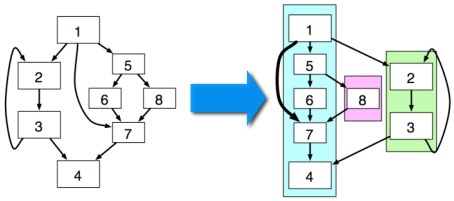
COMPLETED WORK

Trace-Based Allocation

Decompose function into traces of basic blocks

- run simultaneous allocator on each trace
- control flow internal to trace presents difficulty

addressed in proposal document



25

Carnegie Mellon School of Computer Science

COMPLETED WORK

Evaluation

Implemented in gcc 3.4.4 targeting x86

Optimize for code size

- perfect static evaluation
- important metric in its own right

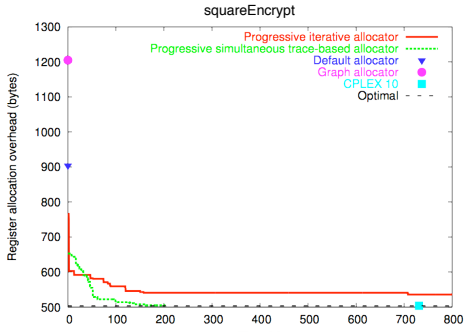
MediaBench, MiBench, Spec95, Spec2000

- over 10,000 functions

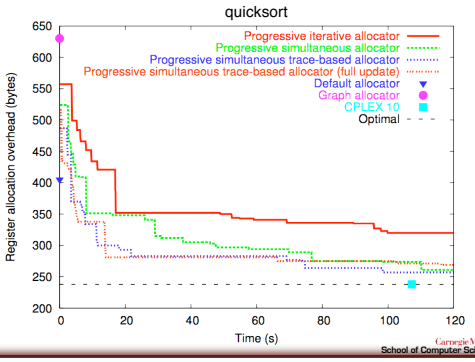
26

Carnegie Mellon School of Computer Science

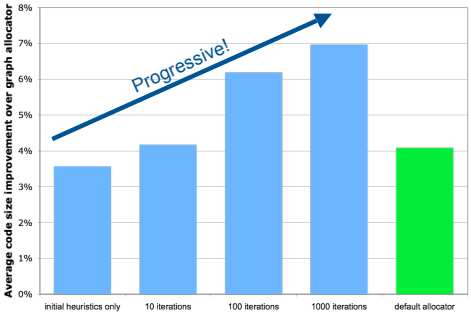
Progressiveness



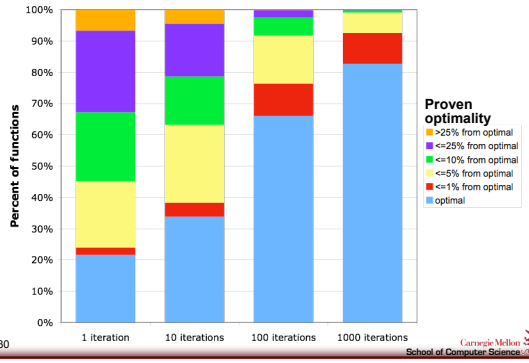
Progressiveness



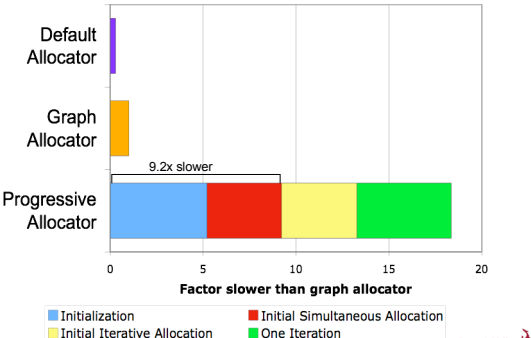
Code Size



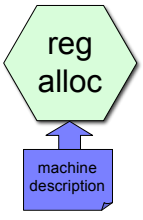
Optimality



Compile Time Slowdown :-(



A Better Register Allocator



– fully utilize machine description
- *explicit* and *expressive* model of costs of allocation for given architecture: **Global MCNF**

– locally optimal
- approach optimality using *progressive* solution technique: **Lagrangian directed allocators**

Outline

I. Motivation
II. Related Work
III. Completed Work
IV. Proposed Work
V. Contributions & Timeline

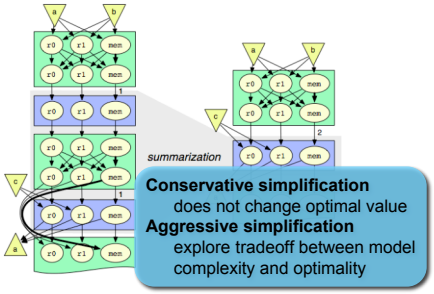
A *Better* Better Register Allocator

Solver Improvements
- Improve initial solution
- Improve quality as prices converge
- Hope to prove approximation bounds

Model Improvements
- Improve accuracy of model
- Model simplification
- Represent uniform register sets efficiently

Model Simplification

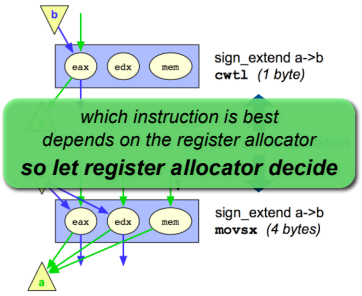
Summarize overly expressive sections of the model



Conservative simplification
does not change optimal value
Aggressive simplification
explore tradeoff between model complexity and optimality

PROPOSED WORK

Instruction Selection Interaction

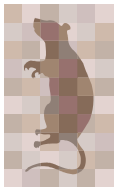


Carnegie Mellon School of Computer Science

PROPOSED WORK

Register Allocation Aware Instruction SElection (RA²ISE)

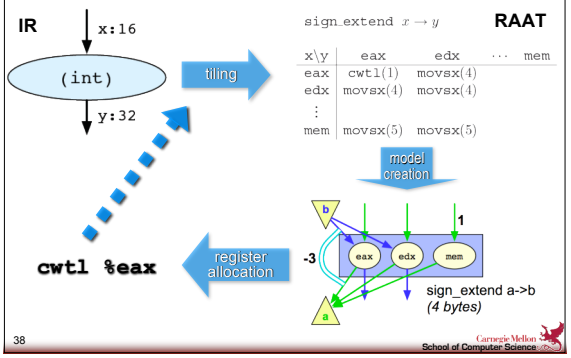
- Instruction selection not finalized until register allocation
- IR tiled with Register Allocation Aware Tiles (RAATs)
- A RAAT represents several instruction sequences
  - different costs
  - a sequence for every possible register allocation



Carnegie Mellon School of Computer Science

PROPOSED WORK

RA²ISE



Carnegie Mellon School of Computer Science

Implementing RA²ISE

Add side-constraints to Global MCNF model

- implement *inter-variable* preferences and constraints
  - "if *x* allocated to r_1 and *y* allocated to r_2, then save three bytes"
  - "*x* and *y* must be allocated to the same register"

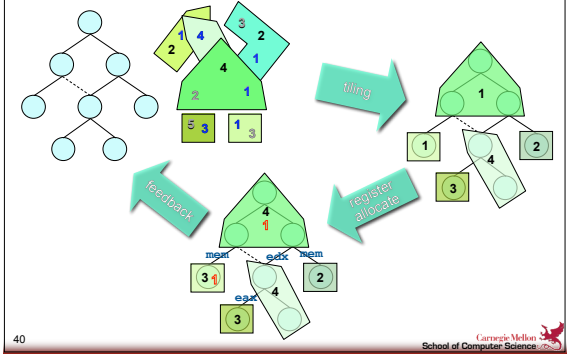
Implement x86 RAATs

- RAAT tables created manually
- GMCNF RAAT representation automatically generated from RAAT table with minimum use of side constraints

Algorithms for tiling RAATs

- leverage existing algorithms
- exploit feedback between passes

Tiling RAATs



Evaluation

Implement in production quality compiler (gcc)

Evaluate code size and simple code speed metric

Evaluate on three different architectures

- x86 (8 registers)
- 68k/ColdFire (16 registers)
- PPC (32 registers)

Outline

I. Motivation
II. Related Work
III. Completed Work
IV. Proposed Work
V. Contributions & Timeline

Contributions

RA²ISE

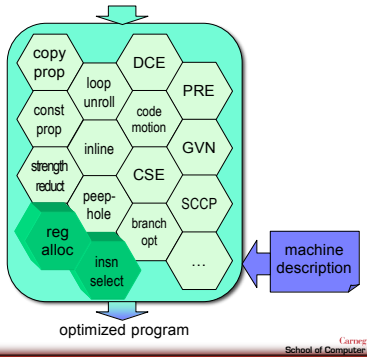
- register allocation aware tiles (RAATs) explicitly encode effect of register allocation on instruction sequence
- algorithms for tiling RAATs
- **expressive model** of register allocation that operates on RAATs and explicitly represents all important components of register allocation
- **progressive solver** for this model that can quickly find decent solution and approaches optimality as more time is allowed for compilation

Comprehensive evaluation of RA²ISE

Thesis Statement

RA²ISE is a principled and effective system for performing instruction selection and register allocation.

One Step Towards a More Principled Compiler



Timeline

Fall 2006	add simple speed metric option to model begin model simplification work improve model accuracy and solver performance
Winter 2006	finish model simplification work add side-constraints to model implement existing gcc tiles as RAATs improve model accuracy and solver performance
Spring 2007	finish implementation of side-constraints and gcc RAATs begin work on RA²ISE infrastructure create gcc-independent set of RAATs for x86 improve model accuracy and solver performance
Summer 2007	finish work on RA²ISE investigate and develop tiling algorithms improve model accuracy and solver performance
Fall 2007	add 68k/ColdFire and PowerPC targets investigate uniform register set simplifications improve model accuracy and solver performance
Winter 2007	begin writing thesis work on improving compile time performance
Spring 2008	finish writing thesis

Questions?