Code Generation

Michael O'Boyle

January, 2014

School of informatics

Overview

- Naive translation and ILOC
- Cost based generation
- Bottom up tiling on low level AST
- Alternative approach based on peephole optimisation
- Superoptimisation
- Multimedia code generation

Introduction

- Aim to generate the most efficient assembly code
- Decouple problem into three phases: Code generation, instruction scheduling, register allocation
- In general NP-complete and strongly interact
- In practise good solutions can be found
- Code generation : would like to automate wherever possible -retargetable ISA specific translation rules plus generic optimiser

Code generation for ILOC

- Mapping IR into assembly code

- ILOC - Simple RISC like instruction set

load r1 − > r2	r2 = Mem[r1]
loadI c1− > r1	r1 = c1
loadAI r1, c1 − > r2	r2 = Mem[r1+c1]
loadA0 r1, r2 − > r3	r3 = Mem[r1+r2]

Similarly for stores

Usual add, sub, mult CMP_LT, cbr

Register copy: i2i r1 − >r2

Cost based translation

- Many ways to do the same thing: i2i r1 $->$ r2, addI r1,0 $->$ r2, lshiftI r1,0 $->$r2 all copy the value of r1 to r2
- If different operators assigned to distinct functional units - big impact
- Simple walk through of first lecture generates inefficient code
- Takes a naive view of location of data and does not exploit different addressing modes available

Example

c * d

c and d are variables located at offsets to global data areas @G and @H

Both are offset by 4

```
loadI @G − > r1
loadI 4 − > r2
loadA0 r1,r2 − > r3
loadI @H − > r4
loadI 4 − > r5
loadA0 r4,r5 − > r6
mult r3,r6 − > r7
```

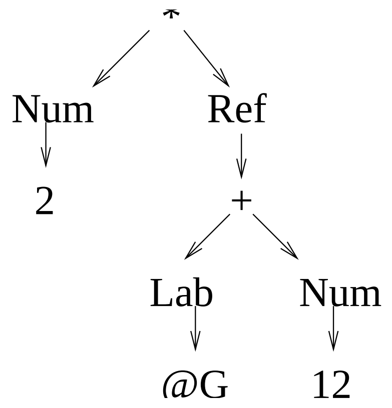
```
loadI 4 − > r1
loadAI r1,@G − > r2
loadAI r1,@H − > r3
mult r2,r3 − > r4
```

Common subexpression hidden from AST

Tree pattern generation

- Represent AST in a low level form exposing storage type of operands
- Tile AST with operation trees generating <ast,op> i.e. op could implement abstract syntax tree ast
- Recursively tile tree and bottom-up select the cheapest tiling - locally optimal.
- Overlaps of trees must match - destination of one tree is the source of another - must agree on storage location - register or memory?
- Operations are connected to AST subtrees by a set of *ambiguous* rewrite rules. Ambiguity allows cost based choice.

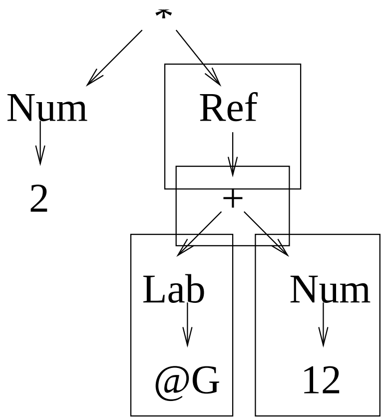
Example: 2 * x. x offset 12 from @G



1:	Reg − > Lab	loadI l − > rnew
2:	Reg − > Num	loadI n − > rnew
3:	Reg− > Ref(Reg)	load r1− >rnew
4:	Reg − > Ref(+(Reg1,Reg2))	loadA0 r1, r2 − > rnew
5:	Reg − > Ref(+(Reg1,num))	loadAI r1, n − > rnew
6:	Reg− > +(Reg1,Reg2)	add r1,r2 − > rnew

Arrows have inverse directions

Tiling using rewrite rules



1: Reg − >Lab1 tiles lower left	loadI @G − > r1
2: Reg− > Num tiles bottom right	loadI 12 − > r2
6: Reg− >+(Reg1,Reg2) tiles +	add r1,r2 − > r3
3: Reg− >Ref(Reg1) tiles REF	load r3 − > r4

Can we do better?

Selecting the best sequence

There are many different sequences available

2	1,5	
3	1,2,4	2,1,4
4	1,2,6,3	2,1,6,3

Selecting lowest cost bottom-up gives

1: Reg $- >$ Lab
5: Reg $- >$ Ref(+(Reg1,num))

loadI @G $- >$ r1
loadAI r1, 12 $- >$ r2

Cost of bottom matching can be reduced using table lookups

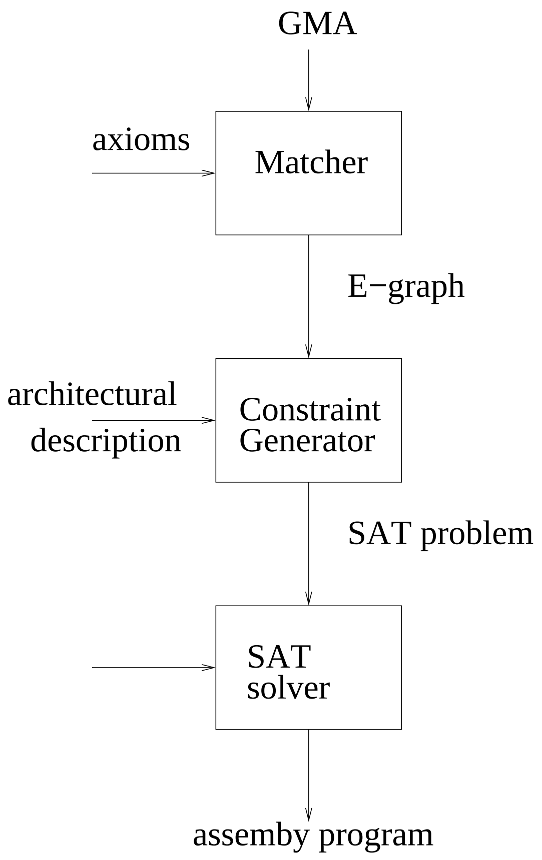
Cost based selection

- Examples assume all operations are equal cost
- Certain ops may be more expensive - divs
- Other approaches available - peephole optimisation
- Works well with linear IR and gives in practise similar performance
- Sensitive to window size - difficult to argue for optimality
- Needs knowledge of when values are dead
- Has difficulty handling general control-flow

Recent work: Denali

- Superoptimiser. Attempt to find optimum code - not just improve.
- Denali A goal directed super-optimizer PLDI 2002 by Joshi, Nelson and Randall. Expect you to read, understand and know this,
- Based on theorem proving over all equivalent programs. Basic idea: use a set of axioms which define equivalent instructions
- Generate a data structure representing all possible equivalent programs. Then use a theorem prover to find the shortest sequence
- “There does not exist a program k cycles or less”. Searches all equivalence to disprove this. Theorem provers designed to be efficient at this type of search

Denali: A goal directed superoptimizer



GMA

axioms

Matcher

E−graph

architectural description

Constraint Generator

SAT problem

SAT solver

assemby program

Denali: A goal directed superoptimizer

Axioms are a mixture of generic and machine specific : Alpha

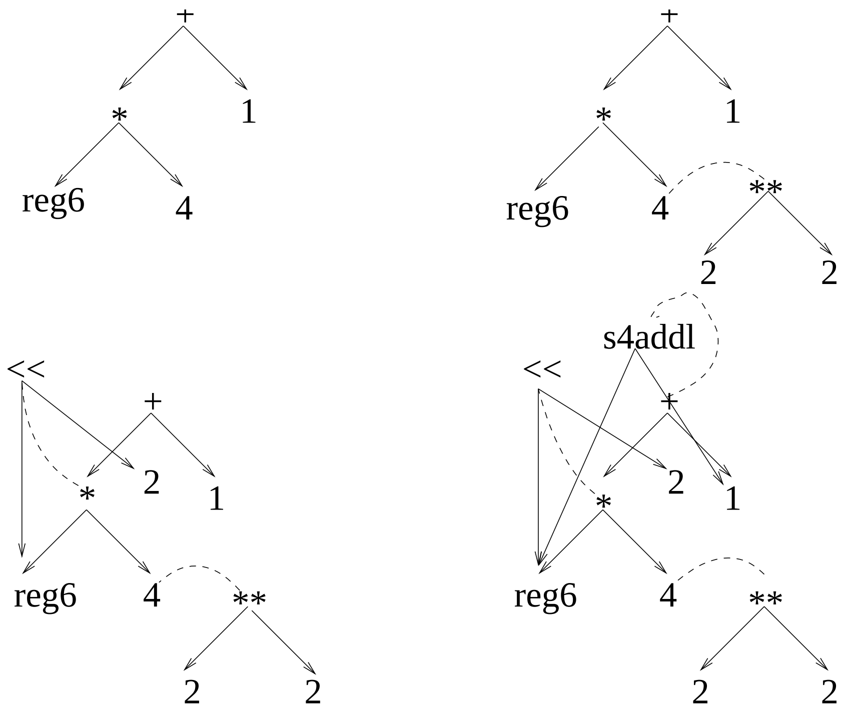
- $4 = 2^2$ – generic
- $(\forall k, n :: k{*}2^n = k{<<}n)$ – machine specific
- $(\forall k, n :: k{*}4{+}n = \mathbf{s4addl}(k, n))$

Equivalences represented in an E-graph.

$O(n)$ graph can represent $O(2^n)$ distinct ways of computing term

Goal: Match expression 1 + reg6 * 4

Denali: E-graph



Dashed lines denote equivalences (matches)

Denali: A goal directed superoptimizer

Once equivalent programs represented, now need to see if there is a solution in K cycles.

Unknowns:

- $L(i, T)$ Term T started at time i
- $A(i, T)$ Term T finished at time i
- $B(i, T)$ Term i finished by time i

Need constraints to solve.

Let $\lambda(T)$ = latency of term T

Denali: A goal directed superoptimizer

Constraints

- $\bigwedge_{i,T}(L(i,T) \Leftrightarrow A(i+\lambda(T)-1,T))$ – arrives λ cycles after being launched
- $\bigwedge_{i,T}\bigwedge_{Q\in args(T)}(L(i,T) \Rightarrow B(i-1,Q))$ –operation cannot be launched till args ready
- $\bigwedge_{Q\in G}(B(K-1,Q))$ – all terms in the goal must be finished within K cycles

Now test with a SAT solver setting K to a suitable number.

Generates excellent code

Finds best code fast. Approximate memory latency, limited implementation

Beyond Denali

- Generating Compiler Optimizations from Proofs, Tate et al POPL 2010
- Uses optimized code examples to abstract optimization
- Generalises by building a proof
- Larger language setting than Denali
- Can then search generalised optimisation space
- Skip the category theory!!
- Stochastic Superoptimization, Schkufza et al ASPLOS 2013.

Multimedia code

- Retargetable code generation key issue in embedded processors
- Heterogeneous instruction sets. Restrictions on function units.
- Exploiting powerful multimedia instructions
- Standard Code generation seems completely blind to parallelism Shorter code may severely restrict ILP
- Denali gets around this but expensive
- Multimedia instructions are often SIMD like. Need parallelisation techniques. Middle section of lectures.

Summary

- Cost based generation
- Bottom up tiling on low level AST
- Alternative approach
- Denali superoptimizer
- Little work on combining this with other phases
- Instruction scheduling next