Compiler Qualification Exam

Terms

- *Interference graph* —
- *Basic block* — a sequence of statements with one program point of entry (at the start of the block) and one point of exit (at the end of the block) . . . i.e. there is no side exists.
- *Super block* — A basic block that allows for side exists.
- *Normal loop* —
- *Back edge* — an edge $a->b$ such that $bDOMa$
- *SSA* — Single static assignment. Each use has only one corresponding def.
- *Extended SSA* —
- *Phi Functions* — a phi function encodes which edges are being entered into the basic block and picks values depending on which edge is entered.
- *Dominator* — a node d dominates n (written $dDOMn$ or $d \gg n$) if every path from the start node to n contains d
- *Immediate Dominator* —
- *Dominance Frontier* —

Presentation

Points-to Analysis in Almost Linear Time (Steensgaard)

Definitions Let a, b, and c be program variables, we define:

- *a points-to b* — there is a statement of the form $a = \&b$ or $a = c$ such that $c = \&b$
- *a aliases b* — there is a variable c such that a points-to c and b points-to c
- *a flows-to c* — c points-to a
- *Flow sensitivity* —
- *Context sensitivity* —
- *Object sensitivity* —
- *Unification* —
- *Heap Modeling* —
- *Modeling Aggregates* —

Main Idea Compute the flow and context insensitive points-to set in linear time. This method was the first to be able to process hundreds of thousands of lines of C code. Compared to Andersen (subset based method) it is less precise.

Algorithm Steensgaard introduces a simple language

```
S ::= x =  y                        // copy y into x
    | x = &amp;y                       // x points y
    | x = *y                        // load y into x
    |*x =  y                        // store y into x
    | x =  op(y...)                 // binary function
    | x =  allocate(y)              // allocate on the heap
    | x =  fun(a...) -> (r...) S*   // function definition
    | x... = p(a...)                // function call with multiple returns
```

Note that this language captures a lot of the essence of pointer behavior in `C`. If one has the following C program for example:

```
int func(int a, int b)
```

He also introduces a simple type system:

$$
\begin{aligned}
&\alpha ::= \tau \times \lambda \\
&\tau ::= \perp \times \mathrm{ref}(\alpha) \\
&\lambda ::= \perp \times \mathrm{lam}(\alpha_1 ... \alpha_n)(\alpha_{n+1} ... \alpha_{n+m})
\end{aligned}
$$

The algorithm is based on unification. Written in datalog (prolog):

```
wellTyped(x = y) = pointsTo().
```

Conclusions

Papers

Data Dependences (High Performance Compilers for Parallel Computing Chapter 5)

Definitions Let S_1 and S_2 be two statements, we define:

- $IN(S)$ — The set of variables used in S_1
- $OUT(S)$ — The set of variables written in Subscript S
- *Flow Dependence* $(S_1 \delta^f S_2)$ — variable written and then used (RAW) ... $OUT(S_1) \cap IN(S_2) \neq \emptyset$
- *Anti-Dependence* $(S_1 \delta^a S_2)$ — variable used and then written (WAR) ... $IN(S_1) \cap OUT(S_2) \neq \emptyset$
- *Output-Dependence* $(S_1 \delta^o S_2)$ — variable written and then written (WAW) ... $OUT(S_1) \cap OUT(S_2) \neq \emptyset$
- *Input Dependence* $(S_1 \delta^i S_2)$ — variable is used and then used ... $IN(S_1) \cap IN(S_2) \neq \emptyset$
- *Dependence* $(S_1 \delta^* S_2)$ — $S_1 \delta^f S_2 \vee S_1 \delta^a S_2 \vee S_1 \delta^o S_2$
- *Address Based Dependence* —
- *Value Based Dependence* —
- *Index Variable Iteration Vector* ($i^{\text{iv}} = \left(\begin{array}{c} i_1 \; i_2 \vdots i_n \end{array} \right)$) —
- *Direction Vector* —
- *Distance Vector* —
- *Iteration Space* —

Main Idea

Algorithm

Conclusions

Data Dependences (High Performance Compilers for Parallel Computing Chapter 9)

Main Idea

Algorithm

Conclusions

A Data Locality Optimizing Algorithm

Main Idea

Algorithm

Conclusions

Parameterized Object Sensitivity for Points-to Analysis for Java

Main Idea

Algorithm

Conclusions

Code generation schema for modulo scheduled loops

Main Idea

Algorithm

Conclusions

An Overview of the PL.8 Compiler

Main Idea Seperation of concerns means that one can develop passes that do not depend on each other — essentially turning the optimization phases into a dataflow sequence.

Algorithm

Conclusions

LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation

Main Idea

Algorithm

Conclusions

Global Data Flow Analysis and Iterative Algorithms

Main Idea

- *Distributive* —
- *Constant Propagation* — not distributive.

Definitions

- *Post order* — vist left child, right child, then root
- *Reverse post order* — reverse order of the post order traversal
- *Reaching Definitions* — a forward may problem

```
gen[n] = {d_v | variable v is defined in BB_n and is not
          followed within n by another defintion v}
kill[n] = {d_v | BB_n contains a defintion of v}

In[n]  = { null if BB_n = start
         { U_{p \in pred} Out[p]
Out[n] = gen[n] U (In[n] \ Kill[n])
```

One can represent this as a lattice with $L = 2^u$ with u being the set of all variables along with their labels generated in the procedure ($variable \times label$). The meet operator $\wedge$ is $\cup$ and $\perp$ is the empty set $\emptyset$ and $\top$ being the set of all expressions u. For a node n the transfer function f_n is $f_n = Gen_{var}[n] \cup (x \cap \overline{Kill_{var}}[n])$

- *Available Expressions* — Forward must problem

```
gen[n] = {d_e | expression e is computed in BB_n and none of its
          uses is redefined}
kill[n] = {d_v | BB_n contains a defintion of v}

In[n]  = { null if BB_n = start
         { \cap_{p \in pred} Out[p]
Out[n] = gen[n] U (In[n] \ Kill[n])
```

One can represent this as a lattice with $L = 2^u$ with u being the set of all expressions computed in the procedure. The meet operator $\wedge$ is $\cap$ and $\perp$ is the empty set $\emptyset$ and $\top$ being the set of all expressions $\underline{u}$. For a node n the transfer function f_n is $f_n = Gen_{expression}[n] \cup (x \cap \overline{Kill_{expression}[n]})$

- *Dominator* — Forward must problem
- *Live Variable* — Backward may problem
- *Very Busy* — Backward must problem
- *Earilest* —
- *Anticipable Expressions* —
- *Def-Use* —
- *Use-Def* —
- *Constant Propagation* —

Algorithm Kam and Ullman introduce a depth-first iterative algorithm

```
In[start] = \bot
for j = 2 to k do
    // if \top \in L use In[j] = \top
    In[j] = /\_{q \in pred*(j)} f_q(In[q])
end
change = true
while change do
    change = false
    for j = 2 to k do // in rPostOrder
        temp = /\_{q \in pred(j)} f_q(In[q])
        if temp != In[j]
            change = true
            In[j] = temp
        end
end
```

With `pred*` defined as `{q | q \in pred(j) and q < j in rPostOrder}`.

Kildall proved that this iterative algorithm converges and computes the maximum fixed point solution. He also showed that `In[n] <= MOP[n]` meaning that the solution is safe and if the transfer function is distributive then `MOP = MFP`. Kam and Ulman showed that if the transfer function is monotone, then `MOP >= MFP`.

In practice it takes a few iterations for this loop to converge.

Conclusions

Lazy Code Motion

Main Idea

Algorithm

Conclusions

Efficiently computing static single assignment form and the control dependence graph

Main Idea

Algorithm

Conclusions

Program Analysis via Graph Reachability

Main Idea Represent data flow as a CFL and use reachability to compute the solution. The following program, for example,

```
func p(g) {
    return g + 1;
}
int x = 1;
int y = 1;
p(x);
p(y);
```

is represented by

```
x = 1 ; y = 1 ; (_p x + 1 )_p (_p y + 1 )_p
```

You can express data flow equations and pointer analysis using CFL reachability.

Algorithm

Conclusions

Exploiting Superword Level Parallelism with Multimedia Instruction Sets

Main Idea Collect chunks of expressions and fuse them to generate vector instructions. For example, if you have the following set of statements:

```
a = x + s
b = y + t
c = z + u
d = w + v
```

then the compiler pass will generate use vectorized add

```
xyzw = float4(x,y,z,w)
stuv = float4(s,t,u,v)
abcd = xyzw + stuv
```

The difficultly happens when you have divergence and have to introduce dummy expressions to faciliate vectorization. The packing/unpacking is also slightly tricky.

Algorithm

Conclusions

Other References

Pointer Analysis: Haven't We Solved This Problem Yet?

Main Idea

Algorithm

Conclusions