Compiler Qualification Exam

Terms

- *Interference graph* — A graph $G = (V, E)$ where V is the set of variables and an edge $v_i \to v_j$ exists iff v_i and v_j's live ranges overlap. The interference graph is usually used to perform register allocation, since a register cannot be used for two different variables live a program point.

- *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. More formally, a sequence of statements $s_0, s_1, \ldots, s_n$ forms a basic block iff s_i dominates s_j if $i > j$ and s_i is not a jump when $i < n$.

- *Super block* — a basic block with side exists allowed. Can be used to optimize program layout (avoid unessary jumps).

- *Normal loop* — a loop with an edge $a \to b$ where the head of the edge dominates its tail (b DOM a)

- *Back edge* — an edge $a \to b$ such that b DOM a

- *SSA* — A property of the IR form such that a virtual register is only assigned once. This implies that there is only one def for each virtual register. It simplifies a lot of analysis. In live range analysis, for example, one needs to look at the preceeding def to find the $def - use$ chain.

- *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 d DOM n or $d \gg n$) if every path from the start node to n contains d. d strictly dominates n if d DOM n and $d \neq n$.

- *Immediate Dominator* —

- *Dominance Frontier* — The dominance frontier of x is the set of all nodes w such that x dominates the predecessor of w and x does not strictly dominate w.

- *Def-Use Chain* — A datastructure consisting of a a definition D of the variable and all uses U of that variable that can reach the use without being killed.

- *Use-Def Chain* — A datastructure consisting of a a use U of the variable and all definitions D of that variable that may reach the use without being killed. Note $d \in Defs(u) iff u \in Uses(d)$. The $Defs$ chain can be computed using reaching definitions and then inverted to compute the $Uses$. For example

  1. if (cond)
  2. `x = ...`
  3. else
  4. `x = ...`
  5. end
  6. ... = x

then $Use - Def(x_6) = S_6 \times S_2, S_4$ since both $defs$ in S_2 and S_4 can reach S_6

Both the $def - use$ and $use - def$ can be computed using data flow analysis. For the $def - use$ set, we can compute the set

```
Let Kill(S_i : x = ...) = {S_i : x}
Let Gen(S_i : ... = x) = {S_i : x}
```

Initialize `Def_IN(BB_i) = {}` and `Def_Out(BB_i) = Gen(BB_i)` and solve the following iteratively:

```
Def_IN(BB_i) = U_{BB_p \in pred(BB_i)} Def_OUT(BB_p)
Def_OUT(BB_i) = Gen(BB_i) U (Def_IN(BB_i) \ Kill(BB_i))
```

This will calculate the `kill` and `gen` set for a basic IR language

```
ClearAll[kill]
kill[Statement[n_, Instruction["Store", {x_, ___}]]] := x
kill[BasicBlock[_, stmts_]] := DeleteDuplicates[kill /@ stmts]
kill[Program[bbs_]] := kill /@ bbs
kill[___] := {}
ClearAll[gen]
gen[Statement[n_, Instruction[_, {_, uses__}]]] := uses
gen[BasicBlock[_, stmts_]] := DeleteDuplicates[gen /@ stmts]
gen[Program[bbs_]] := gen /@ bbs
gen[___] := {}
```

For a given program

```
prog = Program[{
    BasicBlock[1, {
      Statement[1, Instruction["Store", {x, p}]],
      Statement[1, Instruction["Store", {z, p}]]
    }],
    BasicBlock[2, {
      Statement[2, Instruction["Store", {x, q}]]
    }],
    BasicBlock[3, {
      Statement[3, Instruction["Store", {z, x}]]
    }]
}];
```

The `kill` set for the basic blocks are `{{x, z}, {x}, {z}}` and the `gen` set are `{{p}, {q}, {x}}`. Then one can calculate the `in` and `out` sets by finding the fixed points using the above transfer function.

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* —
- *Path 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* ($i^{\text{iv}} = \begin{pmatrix} i_1 \; i_2 \vdots i_n \end{pmatrix}$) —
- *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 Simplify compiler development by introducing a seperation of concerns. This just means that you make each part of the compiler into an independent component that you can debug and optimize seperatly. The downside of seperation of concerns is that you may have to compute a pass more than once.

Algorithm None.

Conclusions 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.

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 Compute where to place the ϕ functions by computing the dominance frontier of the node.

Algorithm A node n dominates m if all paths from the start node to m contain the node n. The dominance graph is composed of

We can compute the dominance graph using a dataflow algorithm with $Dom(start) = \emptyset$ and $Dom(n) =$

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 Construct SLP expressions that can be mapped onto SIMP operations by looking at statements within a basic block and combining them if they use the same operation. Optimizations in the scheduler can be made to avoid packing/unpacking of the data.

Definitions

- *Isomorphic Statements* — are statments that perform the same operations in the same order. The SLP algorithm executes these statments in parallel using a technique called *statement packing*. For example:

  a = b + c * z[i + 0] d = e + f * z[i + 1] r = s + t * z[i + 2] w = x + y * z[i + 3]

can be transformed to

```
{a, d, r, w} = {b, e, s, x} +Simd {c, f, t, y} *Simd {z[i+0], z[i+1], z[i+2], z[i+3]}
```

- A *pack* is an n-tuple, $\langle s_1, \ldots, s_n \rangle$, with $s_1, \ldots, s_n$ are independent isomorphic statements in a basic block.
- A *PackSet* is a set of *packs*.
- A *pair* is a *pack* of size two $\rangle s_{left}, s_{right} \rangle$.
- *Vectorization* is a special case of *SLP* where you try to vectorize the same statement across loop iterations. *SLP* tries to vectorize different statements within the same loop iteration.

Algorithm A high level flow of the transformation is:

1. Unroll loop to transform vector parallelism into SLP
2. Alignment analysis to align each load and store — some architectures do not allow unaligned memory accesses
3. Transform IR into low level form and perform a series of standard compiler optimizations.

The SLP detection/transformation algorithm starts by by looking at independent pairs of statments that contain adjacent memory references. This is done using alignment information and array analysis (in practice nearly every memory reference is adjacent to at most two other references). The statements on the right are transformed into the ones on the left (Identify adjacent memory references).

```
                              UnPacked                 Packed
(1): a = b + c*d[i+0];        (2) : c = 3;             (1) : a = b + c*d[i+0];
(2): c = 3;                   (3) : b = a + c;         (4) : x = y + z*d[i+1];
(3): b = a + c;               (5) : z = 2;
                              (6) : y = x + z;         (4) : x = y + z*d[i+1];
(4): x = y + z*d[i+1];        (8) : u = 1;             (7) : s = t + u*d[i+2];
(5): z = 2;                   (9) : t = s + u;
(6): y = x + z;

(7): s = t + u*d[i+2];
(8): u = 1;
(9): t = s + u;
```

The algorithm then flows the existing $def - use$ chains of existing entries.

```
UnPacked            Packed
(2) : c = 3;        (1) : a = b + c*d[i+0];
                    (4) : x = y + z*d[i+1];
(5) : z = 2;
                    (6) : x = y + z*d[i+1];
(6) : u = 1;        (7) : s = t + u*d[i+2];

                    (3) : b = a + c;
                    (6) : y = x + z;

                    (6) : y = x + z;
                    (9) : t = s + u;
```

The algorithm then flows the existing $use - def$ chains of existing entries.

```
(1) : a = b + c*d[i+0];
(4) : x = y + z*d[i+1];

(6) : x = y + z*d[i+1];
(7) : s = t + u*d[i+2];

(3) : b = a + c;
(6) : y = x + z;

(6) : y = x + z;
(9) : t = s + u;

(2) : c = 3;
(5) : z = 2;
```

```
(5) : z = 2;
(6) : u = 1;
```

The algorithm then merges groups containing the same operations

```
(1) : a = b + c*d[i+0];
(4) : x = y + z*d[i+1];
(7) : s = t + u*d[i+2];

(3) : b = a + c;
(6) : y = x + z;
(9) : t = s + u;

(2) : c = 3;
(5) : z = 2;
(6) : u = 1;
```

The scheduler then looks at the dependence and schedules the operations as SIMD instructions

```
{a, x, s} = {b, y, t} + {c, z, u} * {d[i+0], d[i+1], d[i+2]}
{c, z, u} = {3, 2, 1}
{b, y, t} = {a, x, s} + {c, z, u}
```

Implementation

Conclusions 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.

Packing and unpacking costs may dominate the SIMD operation, the SLP algorithm detects when packed data produced as a result of one computation can be used directly as a source in another computation, hiding some of the packing/unpacking costs.

A Fast Fourier Transform Compiler

Main Idea They mention a few advantages of the special purpose FFTW compiler:

- *Performance* — They are able to generate very performant code that beats hand optimized code. There code is optimal containing over 2400 lines of code including 912 additions and 248 multiplications.
- *Correctness* — Since the algorithm is encoded in a high level language, and the code simplifications are simple algebraic rules, it is easy to prove correctness. In cases where the output was not correct, this was due to a bug in the compiler.
- *Rapid Turnaround* — They are able to modify the compiler and regenerate the entire library in a very short time frame.
- *Effectiveness* — Since the compiler is problem specific, it heavily optimize specific cases. The algebraic simplifications, for example, rely on the DFT being a linear transformation.
- *Derive New Algorithm* — Through a combination of fusing different algorithms for different input sizes, the `genfft` compiler was able to discover new algorithms.

Algorithm

Conclusions

A Comparison of Empirical and Model-Driven Optimization

Main Idea

Algorithm

Conclusions

Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

Main Idea

Algorithm

Conclusions

Trace-based Just-in-Time Type Specialization for Dynamic Languages

Main Idea

Algorithm

Conclusions

Improvements to Graph Coloring Register Allocation

Main Idea

Algorithm

Conclusions

Automatic Generation of Peephole Superoptimizers

Main Idea

Algorithm

Conclusions

Automatic Predicate Abstraction of C Programs

Main Idea

Algorithm

Conclusions

Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability

Main Idea

Algorithm

Conclusions

ABCD: Eliminating Array Bounds Checks on Demand

Main Idea

Algorithm

Conclusions

Other References

References

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

Main Idea

Algorithm

Conclusions