Organization

1. Optimal Representation of Control dependence
   - Definition
   - Is the control dependence graph \( (O(|E|^*|V|) \) space/time) optimal?

2. Our approach:
   - Reduce problem to ROMAN CHARIOTS PROBLEM
   - Build APT data structure in \( O(|E| + |V|) \) space/time
   \( \Rightarrow \) APT is an optimal representation of control dependence

3. Other applications of APT:
   - SSA computation in linear time per variable
   - SDEG computation in linear time per problem
   - DFG computation in linear time per variable

4. Conclusions:
   - APT is a factored form of the CDG
     which requires ‘filtered search’ to answer queries
Queries on Control Dependence Relation:
- **cd(e):** set of nodes control dependent on edge e
- **conds(v):** set of control dependences of node v
- **cdequiv(v):** set of nodes with same control dependences as node v (in same equivalence class as v)

**Application:** program analysis, scheduling for pipelines, parallelization

**Worst-case size of control dependence relation:**

n nested repeat-until loops => size of CDR is \( n(n+3) \)

**The size of the CDR can grow quadratically with program size.**
There have been many unsuccessful efforts to reduce the size of the CDG.

"We therefore conjecture that to enumerate [conds sets] in time proportional to [the size of the set] requires a data structure of quadratic size."

[Cytron,Ferrante,Sarkar, PLDI 1990]

Key Idea (I): Exploit structure of relation

Analogy: Postdominator relation
- queries: immediate postlm of node, all postms of node
- size of relation is $O(|V|^3)$
- relation is transitive, so build transitive reduction (pdom tree)
in $O(|E|)$ time [Harel,Tarjan]
- query time using pdom tree is optimal

=>$\text{There is no point in constructing the entire relation}$

What structure is there in the control dependence relation?

Control dependence relation:
- nodes that are control dependent on an edge $e$
  form a simple path in the postdominator tree
- in a tree, a simple path is uniquely specified by its endpoints

Postdominator tree + endpoints of each control dependence path can be built in $O(|E|)$ space and time

Our Solution:
- reduce control dependence computation to a graph problem called Roman Chariots Problem

- design a data structure called APT (augmented postdominator tree)
  (a) which can be built in $O(|E|)$ space and time, and
  (b) which can be used to answer CD, CONDS and CDEQUIV queries in time proportional to output size.

APT is a data structure for optimal control dependence computation.
CD(n): Which cities are served by chariot n?

Query procedure: (similar to FOW 87)
- Look up entry for chariot n in Route Array (say it is [x,y])
- Traverse nodes in tree T, starting at x and ending at y
- Output all nodes encountered in traversal

(CF. CDG: many routes can share tree nodes/edges)

CD query time is proportional to output size.

Roman Chariots Problem

Given a tree T, and an array A of chariot routes specified by endpoints, design a data structure to answer the following queries in optimal time.
(a) CD(n): Which cities are served by chariot n?
(b) CONDS(w): Which chariots serve city w?
(c) CDEQUIV(w): Which cities are served by the same chariots that serve w?
Key Idea (II): Cache route information in tree

At each node n in the tree, keep a list of chariot #s whose bottom node is n.

Query procedure: CONDS(w)

for each descendant d of w do
  for each route c = [x,y] in list at d do
    if w is a descendant of y
      then output c;
  fi
end

Query time is proportional to # of descendants + size of all lists at descendants

COND(w): Which chariots serve city w?

Query procedure:

for each chariot c in Route Array do
  let route of c be [x,y];
  if w is an ancestor of x
    then output c;
  fi
end

Can we avoid examining all routes in Route Array?

Step 3: Cache route at multiple nodes.

Two extremes:
1) Chariot # stored only at bottom node of route
   Space: O(|V| + |A|)
   Query Time: O(|V| + |Output|)

2) Chariot # stored at all nodes on route
   Space: O(|V| + |A|)
   Query Time: O(|Output|)

Can we have a disciplined caching policy to have linear space and optimal query time?

Refinement: Sort each list by decreasing length.

Query procedure: CONDS(w)

for each descendant d of w do
  for each route c = [x,y] in list at d do
    if w is a descendant of y
      then output c;
    else BREAK; fi
  od
end

At most one “non-overlapping” path is examined at a descendant =>

Query time is proportional to size of output + # of descendants
How do we construct zones?

1. Invariant: For any node \( v \), \(|Z_v| \leq |A_v| + 1\)
   where \( \alpha \) is a design parameter.

   Query time for \( \text{CONDS}(v) = O(|A_v| + |Z_v|) \)
   \( = O((\alpha + 1)|A_v| + 1) \)
   \( = O(|A_v|) \)

II. Build zones bottom-up, making them as large as possible w/o violating invariant

   - \( v \) is a leaf node => make \( v \) a boundary node
   - \( v \) is an interior node =>
     - if \( \sum_{u \in \text{children}(v)} |Z_u| > \alpha |A_v| + 1 \)
       then make \( v \) a boundary node
     - else make \( v \) an interior node

Key idea (III): Cache a route at multiple nodes

Divide tree into ZONES

Query procedure:
Visit only nodes below query node and in the same zone as query node

| Zone construction: For all nodes \( v \), \(|Z_v| \leq \alpha |A_v| + 1| \)
  => Query time \(|A| + |Z| (\alpha + 1)|A_v| |

Caching Rule
- Nodes are partitioned into
  - boundary nodes: lowest nodes in zone
  - interior nodes: all other nodes
- Caching rule:
  - boundary node: store all chariots serving node
  - interior node: store all chariots whose bottom node is that node
- Our algorithm: bottom-up, greedy zone construction
  => space requirements \( \leq |A| + |V| / \alpha \)

\( \alpha = 1 \) (some caching)
\[ \alpha = << \text{(full caching)} \]

**Summary of CONDS Approach:**

- **Query Time:** \((\alpha + 1)|A_v|\)
- **Space:** \(|A| + |V| / \alpha\)

- Parameter \(\alpha\) is used to partition tree into zones
  - \(\ll\): lower query time, increased space requirements
  - \(\gg\): higher query time, lower space requirements
- Nodes are partitioned into
  - boundary nodes: lowest nodes in zone
  - interior nodes: all other nodes
- Caching rule:
  - boundary node: store all chariots serving node
  - interior node: store all chariots whose bottom node is that node
- Query procedure:
  - Visit only nodes below query node and in the same zone as query node
APT

1. Postdominator tree with bidirectional edges
2. dfs-number[v]: integer
   - used for ancestorship determination in CONDS query
3. boundary[v]: boolean
   - true if v is a boundary node, false otherwise
   - used in CONDS query
4. L[v]: list of chariots #s/control dependences
   - boundary node: all chariots serving v (all control dependences of v)
   - interior node: all chariots whose bottom node is v (all immediate control dependences of v)
   - used in CONDS query
5. R[v]: pointer to CDEQUIV equivalence class
   - used in CDEQUIV query

Query time: \((\alpha + 1) \times \text{output-size}\)
Space: \(|E| + |V| / \alpha\)

CDEQUIV(v): Which cities are served by same chariots that serve v?
- Ferrante, Ottenstein, Warren 87: \(O(|E|^3)\) using hashing for set equality
- Cytron, Ferrante, Sarkar 90: \(O(|E|^2)\)
- Ball 92: \(O(|E|)\) for structured programs
- Podgurski 93: \(O(|E|)\) for forward control dependence in general graphs
- Johnson, Pearson, Pingali 94: \(O(|E|)\) for general graphs (optimal)

CDEQUIV for Roman Chariots Problem
- cleaned-up version of JPP94 algorithm
- compute two finger prints for CONDS sets
  - size of CONDS set
  - Lo: lowest node contained in all routes of CONDS set

Two CONDS sets are equal iff they have the same finger-prints.
Can compute finger-prints in \(O(|V| + |A|)\) space and time

Experimental Results
Comparison with factoring:
- Factoring attempts to reduce size of CDG by making nodes ‘share’ control dependences in the representation (CFS 90)

Nodes Edges
Nodes Edges ‘merge’ point

- Our caching approach can be viewed as factoring in which ‘filtered search’ is used to answer queries (Chazelle)

Comparison with factoring:
- Factoring attempts to reduce size of CDG by making nodes ‘share’ control dependences in the representation (CFS 90)

Nodes Edges
Nodes Edges ‘merge’ point

- Our caching approach can be viewed as factoring in which ‘filtered search’ is used to answer queries (Chazelle)

SSA Computation
- phi-placement = iterated dominance frontier computation
- exploit the fact that conds relation is same as edge dominance frontier relation in reverse graph

Solution: Use APT on reverse graph = ADT on CFG

- First, look at DF(S) where S is given offline
  Algorithm: Sort S by level, and query in bottom-up order

Other Applications of APT
Control Dependence
- CONDS
- CDEQUIV
- CD

Dataflow Analysis
- SSA,GSA
- DFG,PDW,VDG,....

ADT : augmented dominator tree (APT on reverse CFG)

ADT and APT
- can be used to build SSA form in O(|E|) per variable
  - subsumes algorithm of Cytron et al (α <=)
  - subsumes algorithm of Sreedhar and Gao (α =>)
- can be used to build DFG in O(|E|) time per variable
  - SESE determination in O(|E|) time
  - see Johnson, Pearson, Pragali (PLDI 94)
  - Johnson’s thesis at Cornell
Algorithm:
- Sort nodes in S by level.
- Remove nodes from sorted list by decreasing level order, and query in ADT
- After a node is queried, mark it in ADT so further queries that reach v do not look below v.

Time = O(|V| + |A|) (O(|E|) in CFG terms)

What if set for querying is given online?
- We can use same strategy provided nodes are presented for querying in bottom-up order.
- Happily, if n is in DF(m), then level(n) <= level(m) !!

=> use a priority queue for ‘dynamic sorting’
- Priority queue implementation: (k = # of keys = height of ADT)
  - van Emde Boas: O(log(log(k))) per insertion and deletion
  - Sreedhar and Gao: use an array of size k
Remarks:

- Time to build SSA form: $O(|E|)$ per variable
- Subsumes algorithms of Cytron et al and Sreedhar and Gao
  $\alpha <<$: Cytron et al. [91] - $O(|E|^2 |V|)$ per variable
  $\alpha >>$: Sreedhar and Gao (PLDI 95) - $O(|E|)$ per variable
- Same idea can be used to build sparse dataflow evaluator graphs
  for other dataflow problems
- What is best value of $\alpha$? Interesting tradeoff
  - small value: repeatedly discover that some node
    is in transitive closure
  - large value: time to compute individual DF sets may be large
  - intermediate value may be best!

Conclusions

1. APT data structure:

   \[
   \begin{align*}
   \text{Query time:} & \quad (\alpha+1) \times \text{output-size} \\
   \text{Preprocessing Space and Time:} & \quad O(|E| + |V|/\alpha)
   \end{align*}
   \]

   Control Dependence   \quad Dataflow Analysis
   CONDS (v): optimal   SSA: $O(|E|)$ per variable
   CDEQUIV(v): optimal   SDEG: $O(|E|)$ per problem
   CD(e): optimal       DFG: $O(|E|)$ per variable

2. Key concepts

   - exploit structure of control dependence relation
   - intelligent caching of information
Applications of Technology

*Top 5 compiler (IBM), Intel... use some of the control dependence algorithms*

*Control dependence algorithms*

*IBM VLIW Compiler: Experiments on the use of dependence flow graph (DFG) as the IF in the VLIW compiler work*

*Ohio State University's Aristotle Analysis System* uses weak control dependence algorithms in Toby compiler (IBM), Intel, etc. Uses some of the control dependence algorithms.

*Digital Continuous Profiling Infrastructure (DCPI): Digital Continuous Profiling Infrastructure uses weak control dependence algorithms*