Single Assignment Compiler, Single Assignment Architecture

Future Gated Single Assignment Form

(Supported by NSF Grant CCF- 1116551)

(Supported by NSF Grant CCF- 1116551)

Shuhan Ding John Earnest Soner Önder

Michigan Technological University

February 18, 2014

- FGSA
- Congruence Classes
- Efficiently Computing FGSA
- Experimental Analysis
- Executing FGSA
- Conclusion

Future Gated Single Assignment

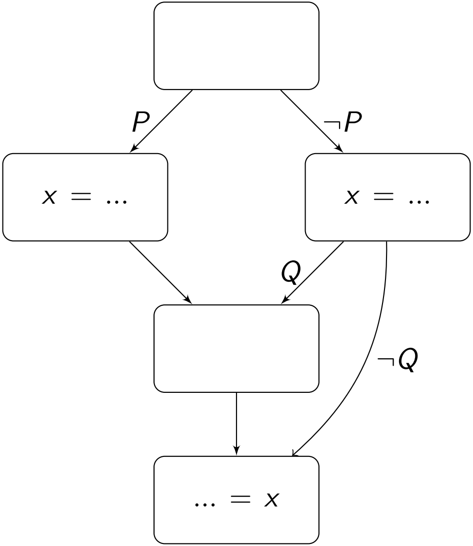
Motivation

- A balance of work must be struck between compilers and microarchitectures
- Close collaboration can simplify both
- A shared program representation can support this

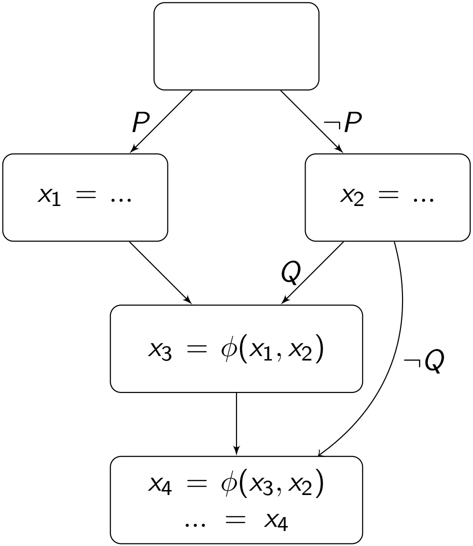
FGSA

- Single-Assignment representation
- Directly usable by optimization algorithms or microarchitectures
- Executable semantics

A Simple CFG



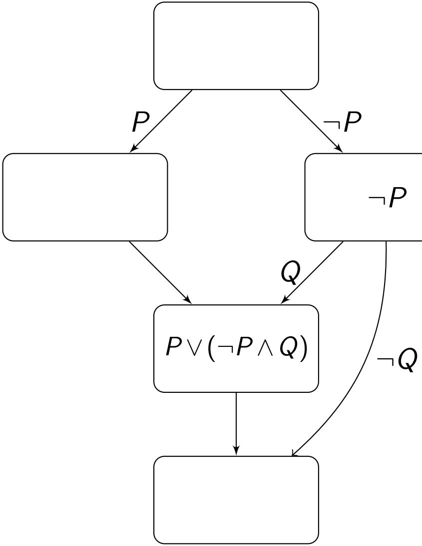
A Simple CFG: SSA



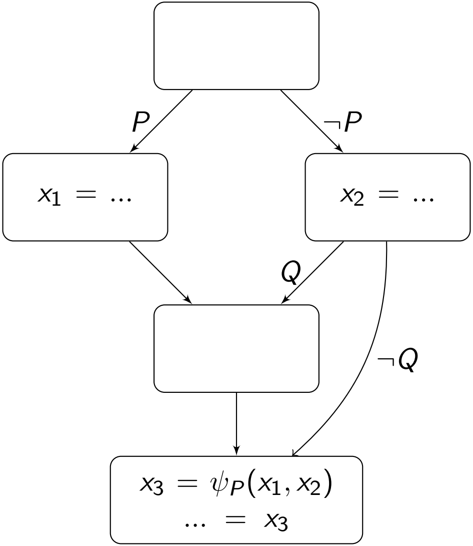
The Predicated Function ψ

$$\psi_{P_1,P_2,\ldots P_n}(x_1, x_2, \ldots x_n)$$

Path Expressions

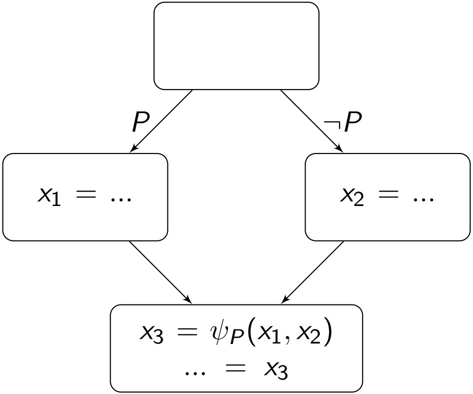


A Simple CFG: FGSA



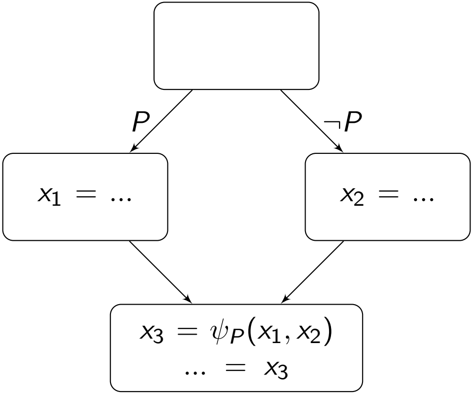
Congruence Classes

Congruence Classes



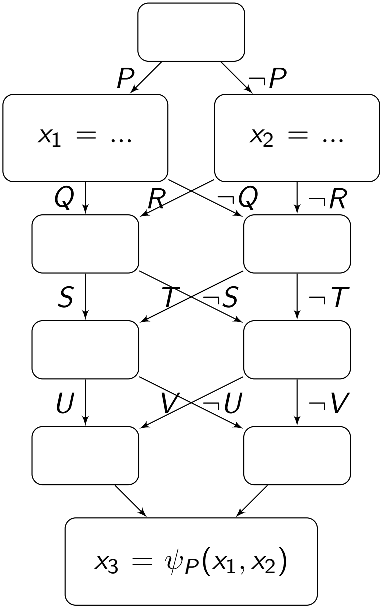
$$\langle D, U \rangle \to \langle \{x_1, x_2\}, \{x_3\} \rangle$$

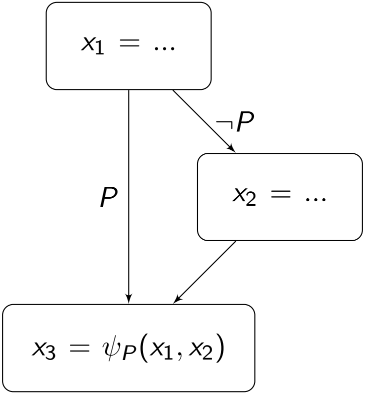
Gated Congruence Classes



$$\langle D, U \rangle_g \rightarrow \langle \{P : x_1, \neg P : x_2\}, \{x_3\} \rangle$$

Minimal Path Expressions for Gating Functions





Theorem 1
Given $CC = \langle\{d_1, d_2\}, U\rangle$ and path expressions p_1 for d_1, p_2 for d_2, the gating predicate expression for d_1 is given by $g_1 = \neg p_2 \wedge p_1$ if there exists a path on which d_2 kills d_1, and $g_1 = p_1$ otherwise.

Efficiently Computing FGSA

Overview

To compute FGSA we find all congruence classes by applying a bidirectional interval analysis algorithm:

1. Scan each block to identify local CCs
2. Process the entire graph by repeatedly applying T1 and T2 transformations until the graph is reduced to a single node
   - As necessary, split irreducible cores using T_R
3. Place gating functions

Local CC computation

$$
\begin{aligned}
... &= x_{u1} \\
x_{d1} &= ... \\
... &= x_{u2} \\
... &= x_{u3} \\
x_{d2} &= ...
\end{aligned}
$$

$$
\begin{aligned}
CC_{u1} &= \langle \emptyset, \{x_{u1}\} \rangle \\
CC_{d1} &= \langle \{x_{d1}\}, \{x_{u2}, x_{u3}\} \rangle \\
CC_{d2} &= \langle \{x_{d2}\}, \emptyset \rangle
\end{aligned}
$$

Perform a backwards linear scan to coalesce together CCs.
CCs which are neither upwards or downwards visible are complete.

Local CC computation

$$CC_{u1} = \langle \emptyset, \{x_{u1}\} \rangle$$

↑ upward visible

$$\dots = x_{u1}$$
$$x_{d2} = \dots$$

$$CC_{d1} = \langle \{x_{d1}\}, \{x_{u2}, x_{u3}\} \rangle$$

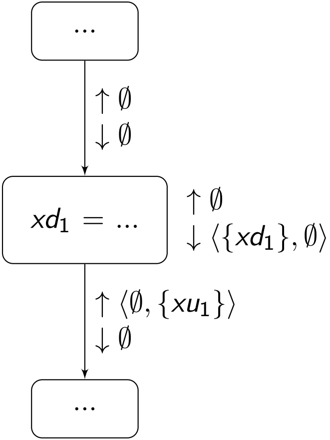
↓ downward visible

$$CC_{d2} = \langle \{x_{d2}\}, \emptyset \rangle$$

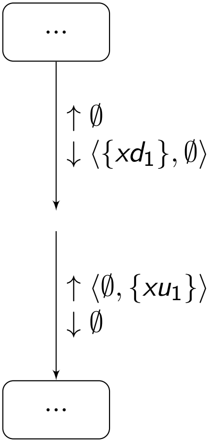
Acyclic Regions and T2

- Candidates for T2 have exactly one predecessor
- The successors of the selected node become successors of the chosen node's predecessors, and edges are chained and merged

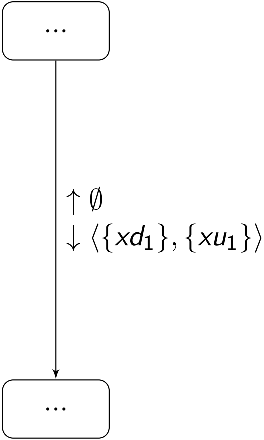
Edge Chaining



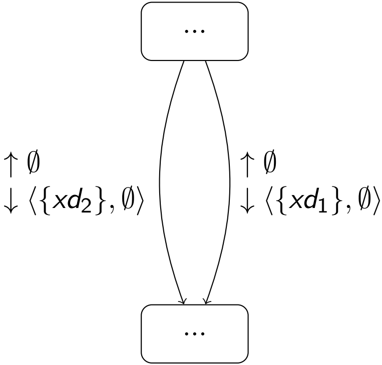
Edge Chaining



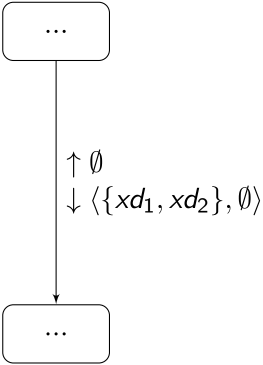
Edge Chaining



Edge Merging



Edge Merging



Cyclic Regions and T1

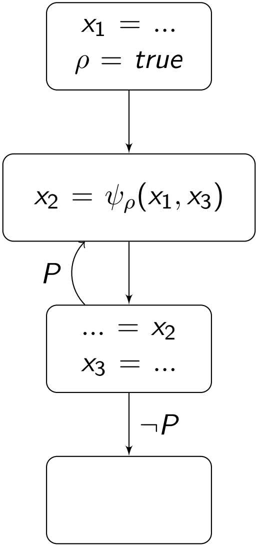
- Candidates for T1 are nodes with a self-pointing back edge
- The back-edge is merged with the node's definitions and as necessary we introduce a gating function guarded by a *read-once predicate* to select from values which flow into the loop and loop-carried values.

Read-Once Predicates

Definition 1
The read-once predicate is a special predicate which becomes false once it is read.

- Used to create gating predicates for cyclic code

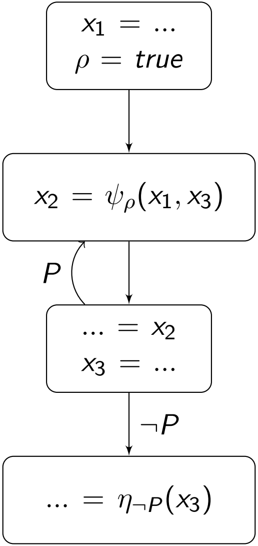
Loop Carried Value



The Exit Function

Definition 2
The *exit* function $\eta(d_i)$ returns the last value of an iteratively executed definition d_i.

Exit Value



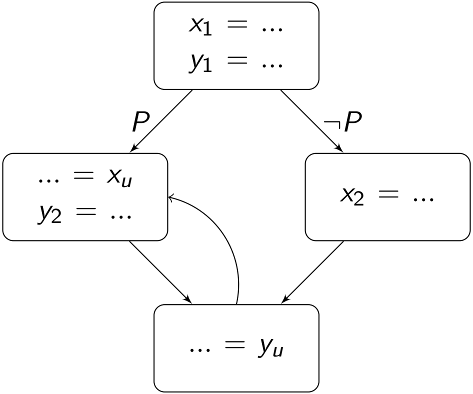
$x_1 = \ldots$
$\rho = true$
$x_2 = \psi_\rho(x_1, x_3)$
P
$\ldots = x_2$
$x_3 = \ldots$
$\neg P$
$\ldots = \eta_{\neg P}(x_3)$

Irreducible Graphs and T_R

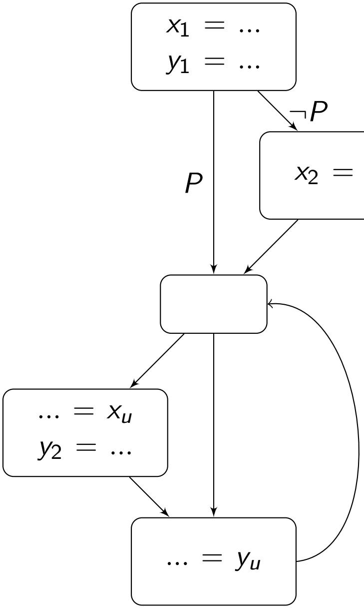
Sometimes we will encounter an irreducible subgraph while performing T1/T2 transformations. In this case, we must convert the graph into a reducible one.

Definition 3
An entrance of an irreducible loop is defined as a node such that there exits a path from the Shared External Dominator (SED) to the node that contains no other nodes in the loop.

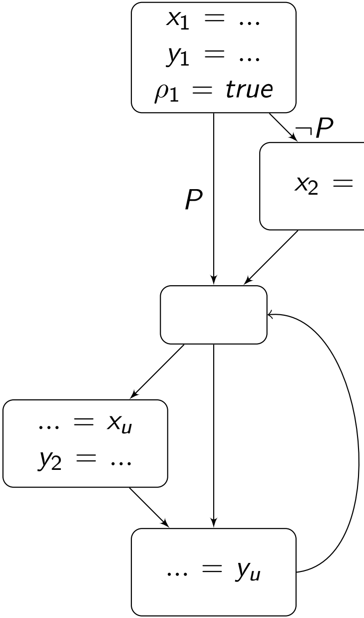
T_R Example



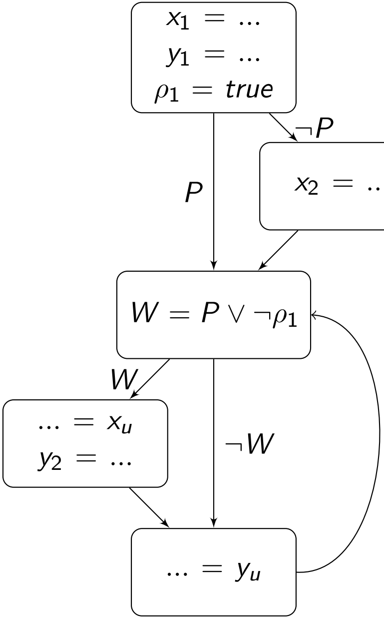
T_R Example



T_R **Example**



T_R Example



Gating Function Construction

- Compute gating predicates from path predicates and reduced reachability information computed during T1/T2
- Gating functions are inserted at the LCDOM node of any uses in the CC
- Definitions which appear below the gating function are marked as a *future value*

Future Values

Definition 4

When instructions i and j are true dependent on each other and the instruction order is reversed, the true dependency becomes a **future value** and is marked on the source operand with the subscript f.

Complexity of FGSA Construction

Given a program, let the number of nodes, edges, user defined variables and instructions be N, E, V and I respectively.

- Local CC computation scans each instruction in each node for each variable. Thus, time complexity per variable is $\frac{O(I)}{V}$
- During CC propagation edge-chaining runs for each node with a single predecessor (O(N)), edge-merging runs over edges in the graph (O(E)) and runtime for T1 is bounded by O(N)
- For each CC definition (O(N) CCs containing O(N) definitions each as a loose bound), we must query the reduced reachable sets some number of times $\sum_{CC_i} |CC_i.D|$

Loose bound for time complexity is $\frac{O(I)}{V} + O(N + E) + O(N^2)$
Expected overall time complexity is $\frac{O(I)}{V} + O(N + E)$

Experimental Analysis

Methodology

- Compute the number of gated CCs and compare with the number of ϕ functions constructed in SSA
- SPEC CINT2000 test suite with *-O3* optimizations
- GCC generates SSA via Cytron's Algorithm
  - Tested with and without ϕ-pruning
- Data collected per function in each benchmark

Summary

- Comparing CCs with pruned ϕs, we observe a maximum reduction of 67.5% from a function in `186.crafty` and an average reduction of 7.7%
- CCs consisting of two definitions are dominant, accounting for at least 62% in all the benchmarks
- CCs consisting of more than four definitions account for $\leq$ 13.38% in worst-case benchmarks
- Median predicate expression length in the whole suite is ≤ 2
- Predicate expressions longer than eight elements make up $<$ 10% of the CCs

Executing FGSA

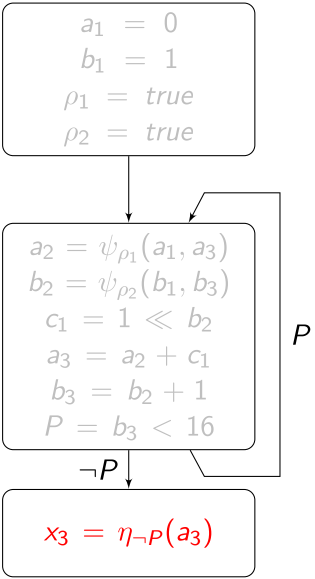
Executing FGSA

- Traditional architectures (via inverse transformation)
- Control-flow architectures supporting future values
- Demand-driven architectures...

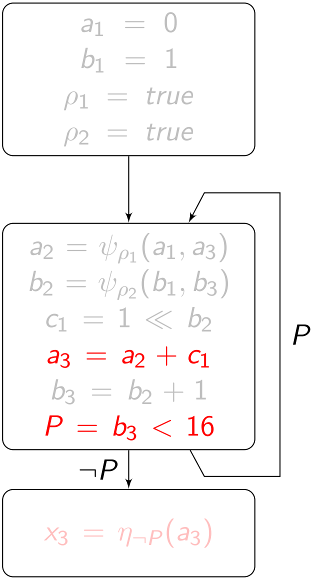
Demand-Driven Interpretation

```
int a = 0;
for(int b = 1; b < 16; b++) {
    a += 1 << b;
}
... = a;
```

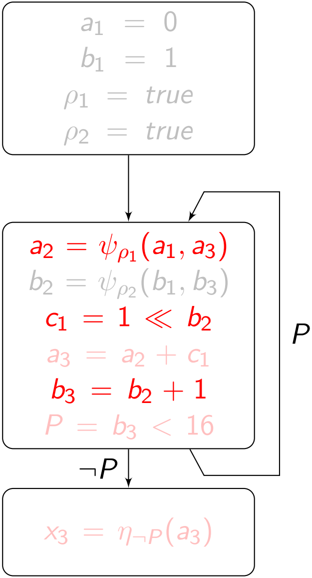
Demand-Driven Interpretation



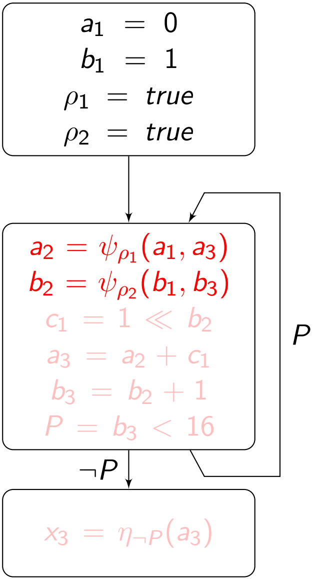
Demand-Driven Interpretation



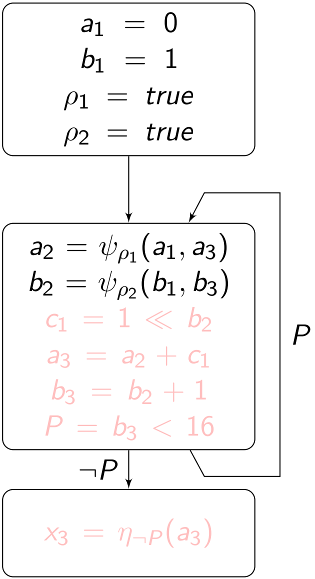
Demand-Driven Interpretation



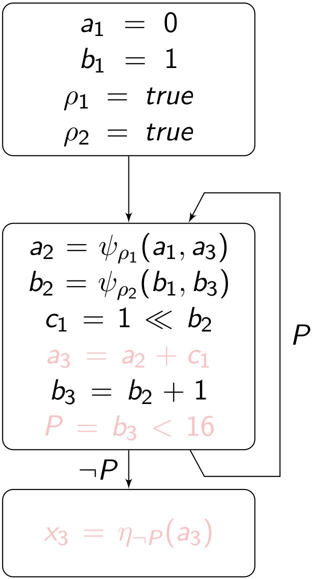
Demand-Driven Interpretation



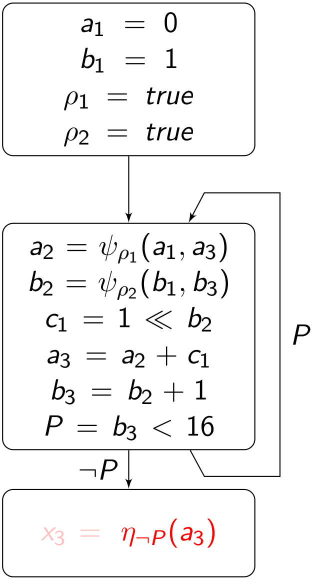
Demand-Driven Interpretation



Demand-Driven Interpretation



Demand-Driven Interpretation



Conclusion

Overview of FGSA

- A static-single-assignment IR with executable semantics
- Densely represents use-def relationships with gated congruence classes
- Can be efficiently computed using a series of T1/T2 transformations
- Construction handles irreducible graphs without exponential code expansion
- Convenient both for optimization and direct execution by hardware

Future Work

- Formal analysis, adaptation and implementation of well-known optimizations using this representation
- Development of micro-architectures that take advantage of FGSA
- Exploration of alternative forms of execution under this paradigm

Questions?

CCs vs ϕ-functions over REAL.

	vars	phis	ccs	% Reduction Max	Average
164.gzip	3715	624	514	42.86	8.85
	3715	4401	514	100	69.76
175.vpr	16648	1309	1092	61.11	7.39
	16648	15773	1092	100	81.26
176.gcc	125212	15810	14206	66.67	4.8
	125212	152079	14206	100	72.98
181.mcf	899	161	117	60	12.17
	899	666	117	100	63.92
186.crafy	14341	1485	1226	67.47	10.55
	14341	15972	1226	100	79.77
197.parser	18720	2887	2653	50	6.08
	18720	25656	2653	100	60.59
253.perlbmk	20330	1789	1656	50	2.83
	20330	16578	1656	100	77.97
255.vortex	36585	1913	1747	50	1.9
	36585	16151	1747	100	77.97
256.bzip2	3598	342	286	50	12
	3598	2421	286	100	71.21
300.twolf	21676	2653	1991	64.91	10.22
	21676	34162	1991	100	81.18

Number of definitions in CCs

	ccs	2defs%	3defs%	4defs%	4^{+}defs%
164.gzip	514	78.79	11.87	4.28	5.06
175.vpr	1092	81.32	7.97	7.97	2.75
176.gcc	14206	76.95	10.14	4.65	8.26
181.mcf	117	68.38	27.35	1.71	2.56
186.crafy	1226	62.07	14.52	10.03	13.38
197.parser	2653	79.80	16.66	2.41	1.13
253.perlbmk	1656	79.71	8.33	7.13	4.83
255.vortex	1747	87.58	5.15	3.15	4.12
256.bzip2	286	80.42	12.24	5.59	1.75
300.twolf	1991	76.49	10.90	9.94	2.66

Length of CC Predicate Expressions

Benchmark	median	average	$\% > 4$	$\% > 8$	max
164.gzip	1	1.98	12.5	0.4	13
175.vpr	1	2.06	7.1	1.4	31
176.gcc	2	3.79	20.3	9.2	132
181.mcf	1	1.97	6.0	1.7	9
186.crafty	2	3.15	16.7	6.1	95
197.parser	2	2.27	12.9	1.3	83
253.perlbmk	1	2.50	12.6	5.3	31
255.vortex	1	2.01	11.2	3.4	17
256.bzip2	1	1.71	4.6	1.4	15
300.twolf	1	2.23	8.1	3.5	32