Large Spurious Cycles in Global Static Analyses and Their Algorithmic Mitigation

Hakjoo Oh
pronto@ropas.snu.ac.kr

July 10, 2009
ROSAEC Workshop

Motivation

Airac

- Sparrow's buffer-overrun-analysis engine
- an interval-domain-based abstract interpreter
- our platform for domain-specific value analyzer
  - e.g., domain-specific analyzer for JP Morgan's CDS software

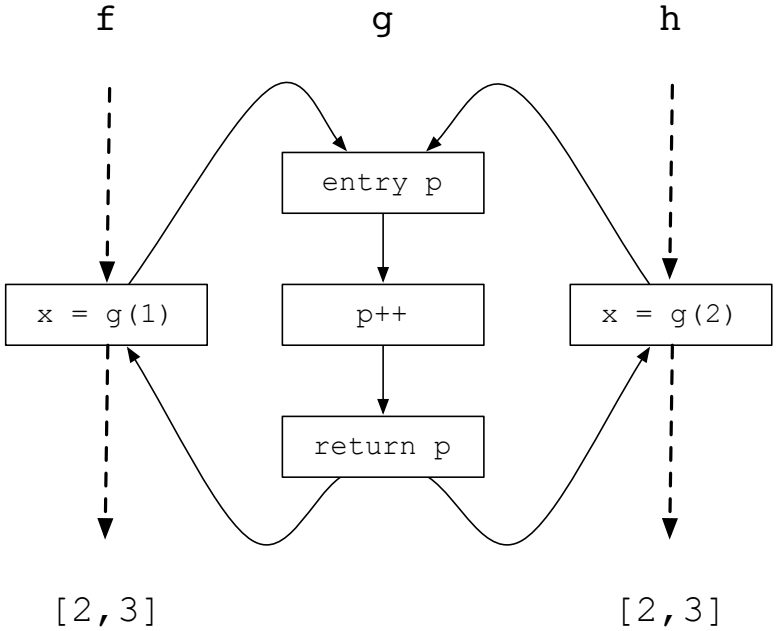
is too slow.

Program	LOC	#Basic-Blocks	Time(s) [1]	#Iterations
gzip-1.2.4a	7,327	6,541	4601.23	653,063
bc-1.06	13,093	9,298	23515.27	1,964,396
less-290	18,449	7,754	46274.67	3,149,284
tar-1.13	20,258	10,800	75013.88	4,748,749
make-3.76.1	27,304	11,061	88221.06	4,613,382

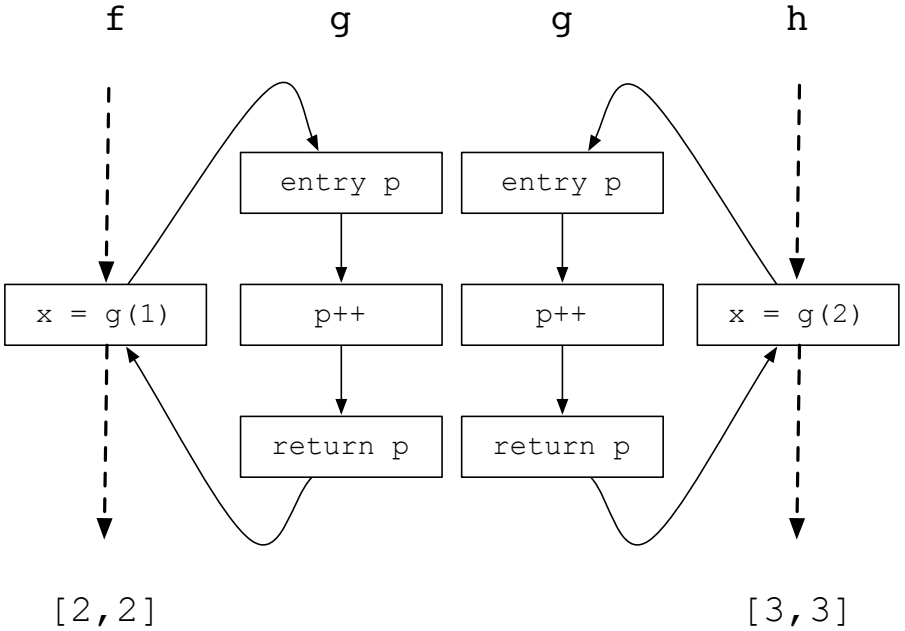
We identify a problem (due to lack of enough context-sensitivity) and present its solution.

[1] Pentium4 3.2GHz, 4GB RAM

Spurious Paths & Context-sensitivity



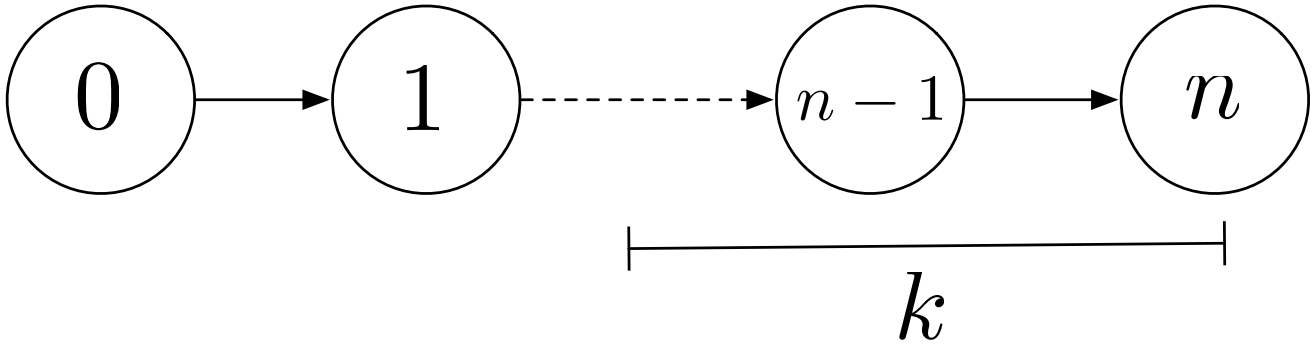
context-insensitivity



context-sensitivity

Spurious Paths Are Inevitable in Practice

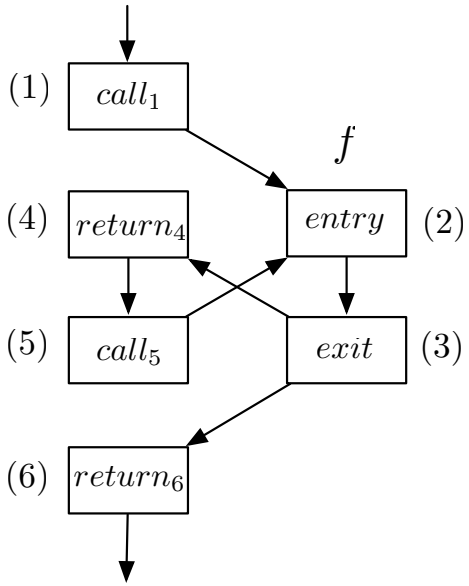
- Exponential (or infinite) number of contexts in programs.
- In static analysis, the number of distinguished contexts must be finite.
  - k-limiting (k-CFA, k-suffix)
- k-limiting quickly hits a limit in practice.



Any k-limiting context-sensitivity inevitably has spurious paths.

Spurious Paths Make Cycles

```
int f() { ... }
int main() {
   f();
   f();
}
```



Spurious cycles degrade both precision and speed, because

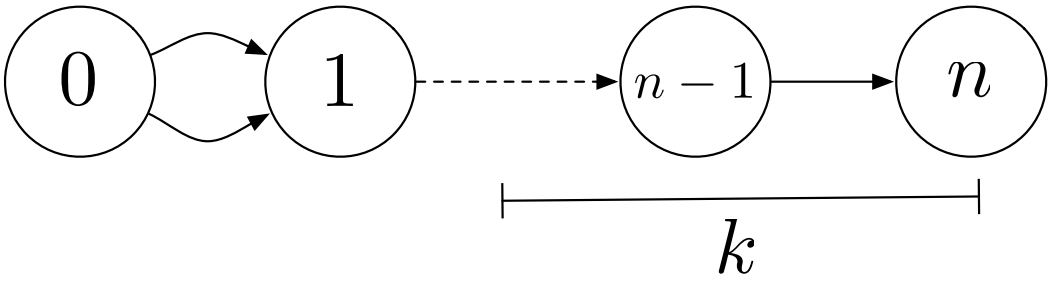
- they model spurious information flow.
- solving cyclic equations is repeatedly applying the equations in vain until a fixpoint is reached.

Spurious Cycles in Reality

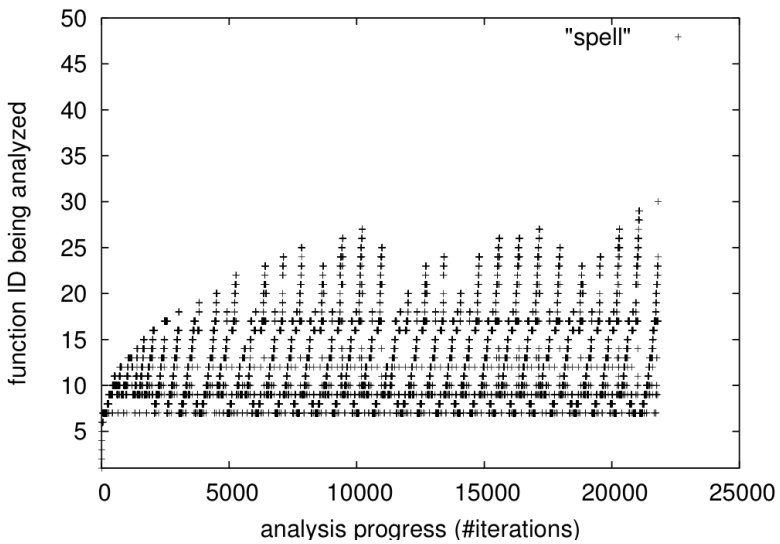
In real programs, a single large spurious cycle spans almost all parts of the program.

Program	Procedures in the largest cycle	Basic-blocks in the largest cycle
spell-1.0	24/31(77%)	751/782(95%)
gzip-1.2.4a	100/135(74%)	5,988/6,271(95%)
sed-4.0.8	230/294(78%)	14,559/14,976(97%)
tar-1.13	205/222(92%)	10,194/10,800(94%)
wget-1.9	346/434(80%)	15,249/16,544(92%)
bison-1.875	410/832(49%)	12,558/18,110(69%)
proftpd-1.3.1	940/1,096(85%)	35,386/41,062(86%)
apache-2.2.2	1,364/2,075(66%)	71,719/95,179(75%)

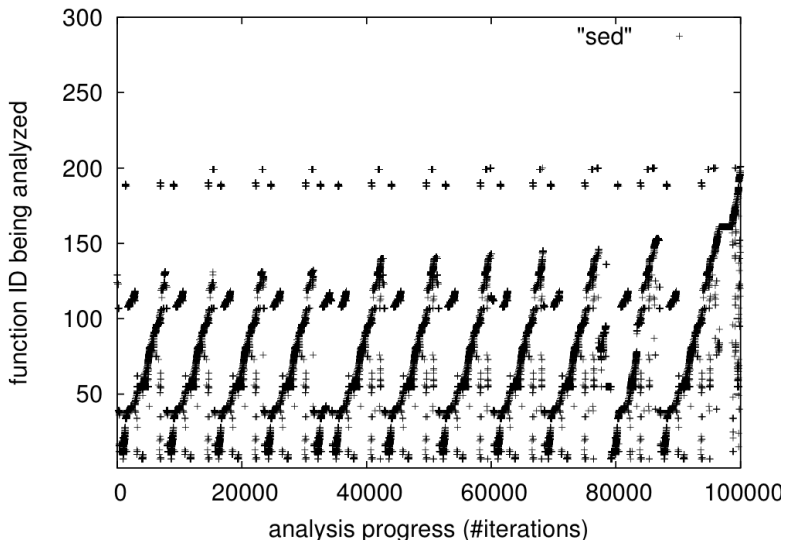
k-limiting is not much effective to mitigate the problem.



Effects on Analysis



spell-1.0 (total #procs:31)
(2,213 LOC, > 30 repetitions)



sed-4.0.8 (total #procs:294)
(26,807 LOC, > 150 repetitions)

Related Work

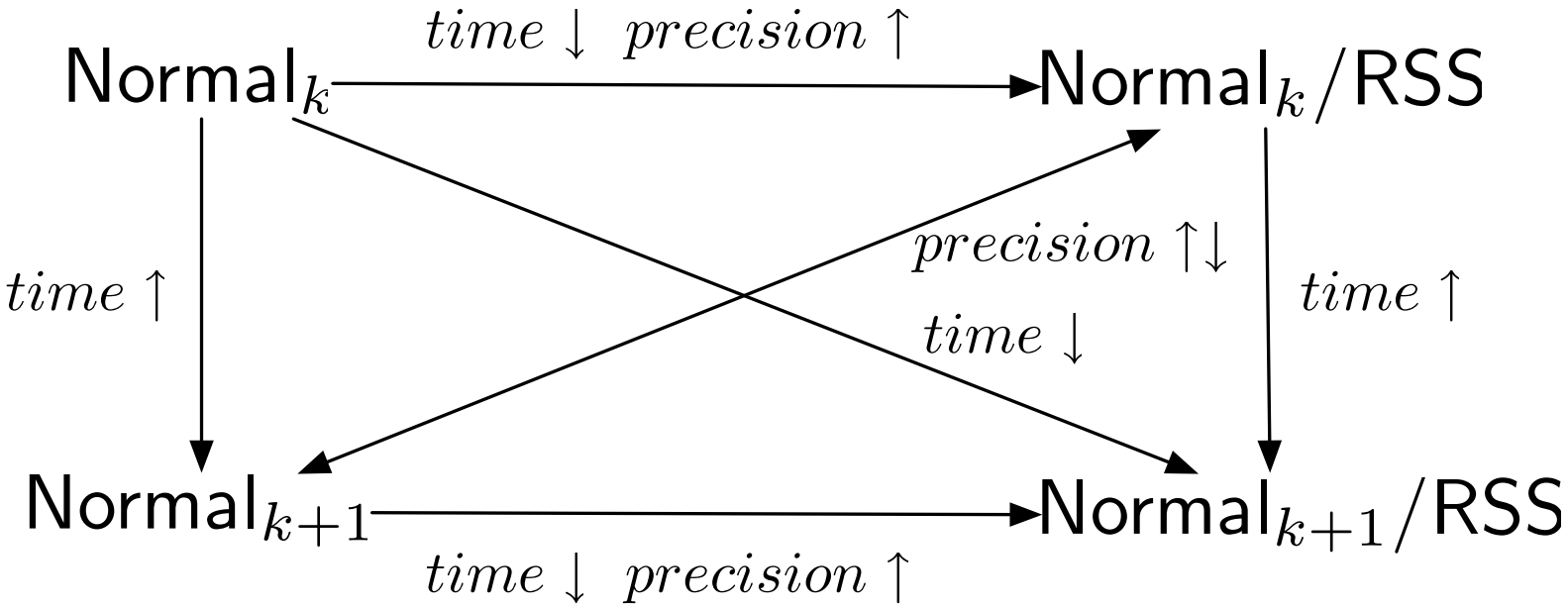
- Lots of work that eliminate spurious paths.
  - Especially, in data flow analysis.
  - Functional approaches: elimination, iterative, RHS
  - Full call-strings approaches

  But they are limited to restricted classes of analysis problems.
- k-limiting is one of very few options available for more general analyses problems[2] to mitigate spurious paths.
  - Our technique is a simple extension to it, which improves both speed and precision.

[2] e.g., infinite domain with non-distributive flow functions

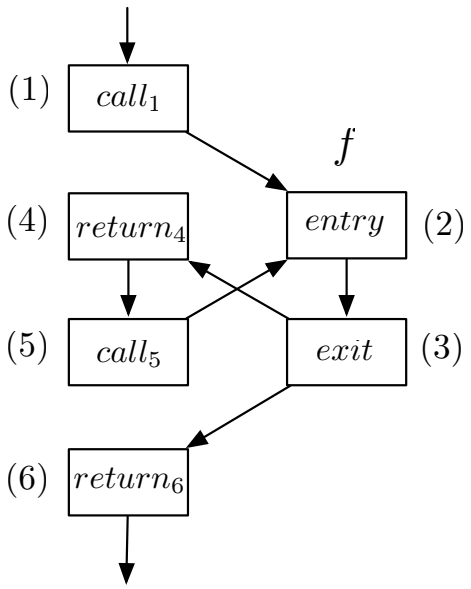
Benefits of Our Extension

- Normal_k: the classical k-limiting algorithm
- $\text{Normal}_k/\text{RSS}$: the k-limiting with our extension



Basic Idea

Change the worklist ordering so that multiple returns are unnecessary.



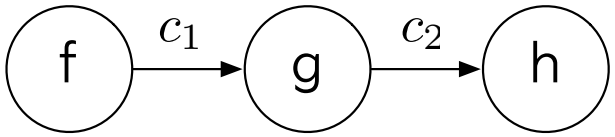
$\cdots \rightarrow (5) \rightarrow (2) \rightarrow (3)$

- Normal$_k$ inserts (4) and (6).
- But, the current analysis of f is not related to (4), but only to (6).
- Exclusive analysis of the callee is key.

One-call-per-procedure, exclusively

Prioritizing a callee over call-sites that invoke the callee

For example,



$W = \{n_h, c_1, c_2, \cdots\}$

- prioritize n_h over c_1 and c_2, and select one from $\{n_h, \cdots\}$

Normal_k/RSS Algorithm

$\mathit{FixpointIterate}\ (\mathcal{W}, \mathcal{T}) =$

$\mathit{ReturnSite} := \emptyset$

repeat

$\quad \mathcal{S} := \{(\mathit{call}_f^{g,r}, \_) \in \mathcal{W} \mid (n_h, \_) \in \mathcal{W} \wedge \text{reach}(g, h) \wedge \neg\text{recursive}(g)\}$

$\quad (n, \delta) := \text{choose}(\ \mathcal{W} \setminus \mathcal{S}\ )$

$\quad m := \hat{\mathcal{F}}\ n\ (\mathcal{T}(n)(\delta))$

$\quad$ **if** $n = \mathit{call}_f^{g,r} \wedge \neg\text{recursive}(g)$ **then** $\mathit{ReturnSite}(g) := (r, \delta)$

$\quad$ **if** $n = \mathit{exit}_g \wedge \neg\text{recursive}(g)$ **then**

$\quad\quad (r, \delta_r) := \mathit{ReturnSite}(g)$

$\quad\quad$ **if** $m \not\sqsubseteq \mathcal{T}(r)(\delta_r)$

$\quad\quad\quad \mathcal{W} := \mathcal{W} \cup \{(r, \delta_r)\}$

$\quad\quad\quad \mathcal{T}(r)(\delta_r) := \mathcal{T}(r)(\delta_r) \sqcup m$

$\quad$ **else**

$\quad\quad$ **for all** $(n', \delta') \in \mathcal{N}(n, \delta)$ **do**

$\quad\quad\quad$ **if** $m \not\sqsubseteq \mathcal{T}(n')(\delta')$

$\quad\quad\quad\quad \mathcal{W} := \mathcal{W} \cup \{(n', \delta')\}$

$\quad\quad\quad\quad \mathcal{T}(n')(\delta') := \mathcal{T}(n')(\delta') \sqcup m$

until $\mathcal{W} = \emptyset$

Purely Algorithmic Technique

Because it is purely an algorithmic technique inside the worklist-based fixpoint iteration routine,

- it is orthogonal to the performance gains of context-sensitive abstract semantics.
- the correctness is obvious enough to avoid the burden of a safety proof of otherwise newly designed abstract semantics.

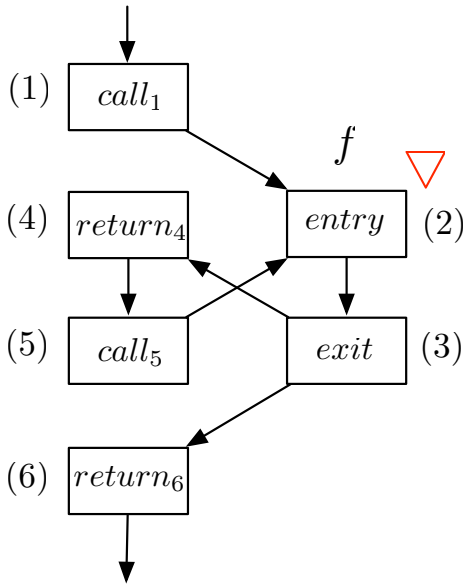
Correctness & Precision

- The result is not a fixpoint of the given equation system, but still a sound approximation of program semantics.
- Our algorithm is always at least as precise as the original worklist algorithm.
  - by removing unnecessary computations (analyzing more means losing precision)

More Precision: Less Widening Points

We no more apply widenings to interprocedural loopheads created by non-recursive procedure calls.

```
int g = 0;
int f() { g++; }
int main() {
   f();
   f();
}
```



- Normal worklist algorithm computes $g = [0, +\infty]$
- Our algorithm computes $g = [0, 2]$

Experiments

We compare speed and precision between Normal_k/RSS and Normal_k.

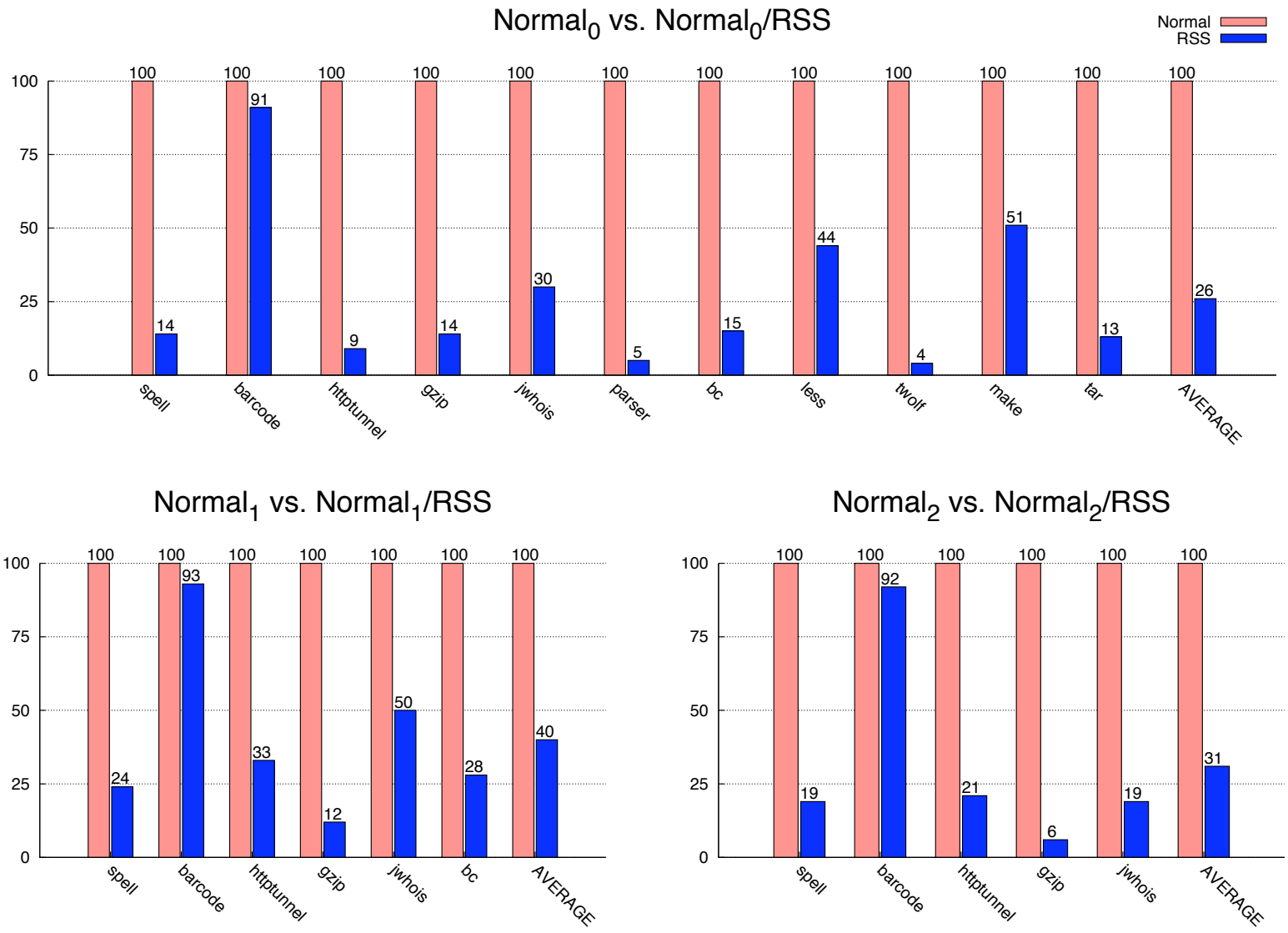
- Normal_k: Sparrow's underlying worklist algorithm
- Normal_k/RSS: Normal_k with our algorithmic technique

Benchmark Programs

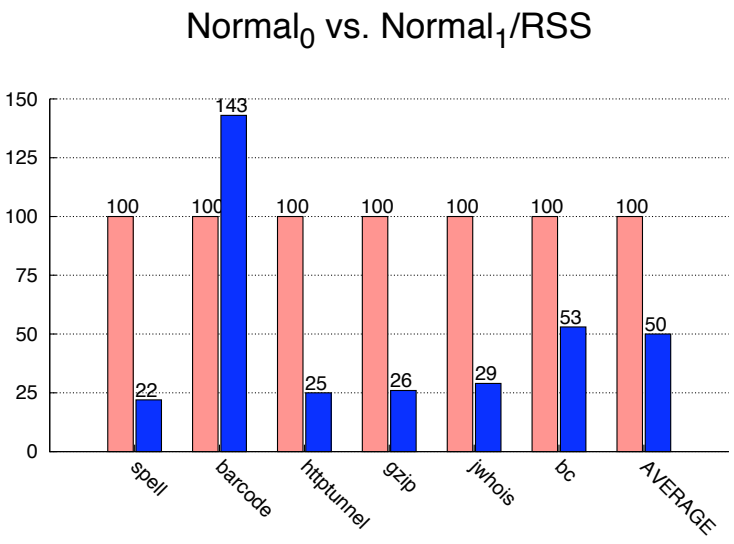
11 open-source software packages.

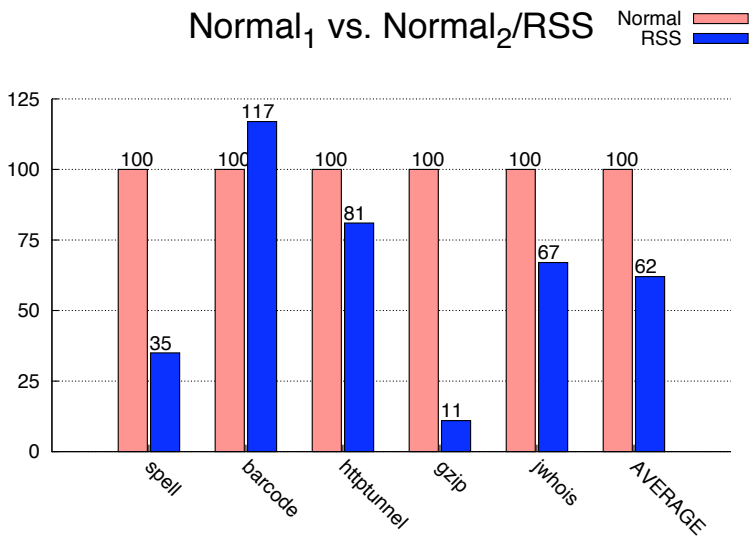
Program	LOC	#Basic-Blocks
spell-1.0	2,213	782
barcode-0.96	4,460	2,634
httptunnel-3.3	6,174	2,757
gzip-1.2.4a	7,327	6,271
jwhois-3.0.1	9,344	5,147
parser	10,900	9,298
bc-1.06	13,093	4,924
less-290	18,449	7,754
twolf	19,700	14,610
tar-1.13	20,258	10,800
make-3.76.1	27,304	11,061

Comparison of Speed between Normal_k and $\text{Normal}_k/\text{RSS}$



Comparison of Speed between Normal_k vs. Normal_{k+1}/RSS





Comparison of Precision between Normal_0 and $\text{Normal}_0/\text{RSS}$

Program	**Analysis**	#intervals in the joined memory			
		$[c, c]$	$[c_1, c_2]$	$[c, +\infty)$	$(-\infty, +\infty)$
spell	Normal_0	345	88	33	143
	$\text{Normal}_0/\text{RSS}$	345	89	35	140
barcode	Normal_0	2136	588	240	527
	$\text{Normal}_0/\text{RSS}$	2136	589	240	526
httptunnel	Normal_0	1337	342	120	481
	$\text{Normal}_0/\text{RSS}$	1345	342	120	473
gzip	Normal_0	1995	714	255	1214
	$\text{Normal}_0/\text{RSS}$	1995	716	255	1212
jwhois	Normal_0	2740	415	961	1036
	$\text{Normal}_0/\text{RSS}$	2740	415	961	1036

Conclusion

We suggest the following implementation guideline:

- When Normal_k hits the memory cost limit: then use $\text{Normal}_k/\text{RSS}$ instead.
  - $\text{Normal}_k/\text{RSS}$ is empirically faster than Normal_k
  - $\text{Normal}_k/\text{RSS}$ is at least as precise as Normal_k
  - $\text{Normal}_k/\text{RSS}$ requires in addition just as many memory entities as the number of procedures.
- When Normal_k hits the time cost limit: then, if memory permits, consider using $\text{Normal}_{k+1}/\text{RSS}$ instead.
  - $\text{Normal}_{k+1}/\text{RSS}$ can be even faster than Normal_k
  - $\text{Normal}_{k+1}/\text{RSS}$ is in principle more precise than Normal_k
  - it requires in addition just as many entities as the number of procedures than Normal_{k+1}