Large Spurious Cycle in Global Static Analyses and Its Algorithmic Mitigation

Hakjoo Oh
pronto@ropas.snu.ac.kr

School of Computer Science and Engineering
Seoul National University
Korea

Dec 14, 2009
Asian Symposium on Programming Language and Systems

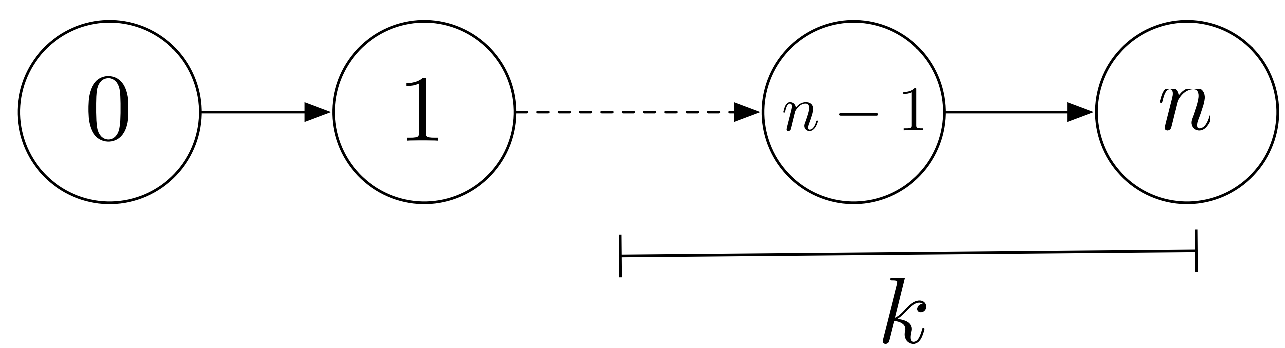
Goal

A performance problem is identified and solved.

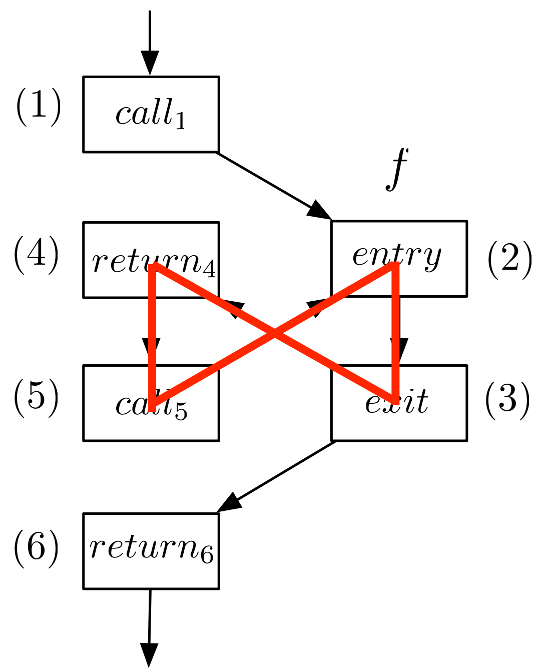
$\overset{time \downarrow \;\; precision \uparrow}{\text{Normal}_k \longrightarrow \text{RSS}_k}$

An extension of the classical call-strings approach

Conventional context-sensitive analysis, distinguishing the last *k* call-sites to each procedure (k-limiting).



Spurious Cycle in Static Analysis

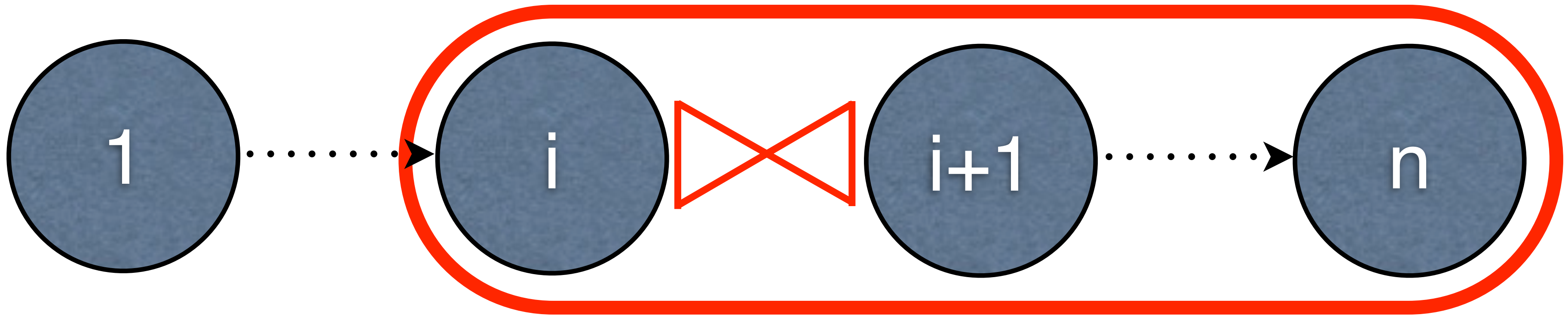


$$
\begin{aligned}
call_1 &= \cdots & \cdots\ (1)\\
entry &= call_1 \sqcup call_5 & \cdots\ (2)\\
exit &= \hat{\mathcal{F}}_{body_f}(entry) & \cdots\ (3)\\
return_4 &= exit & \cdots\ (4)\\
call_5 &= return_4 & \cdots\ (5)\\
return_6 &= exit & \cdots\ (6)
\end{aligned}
$$

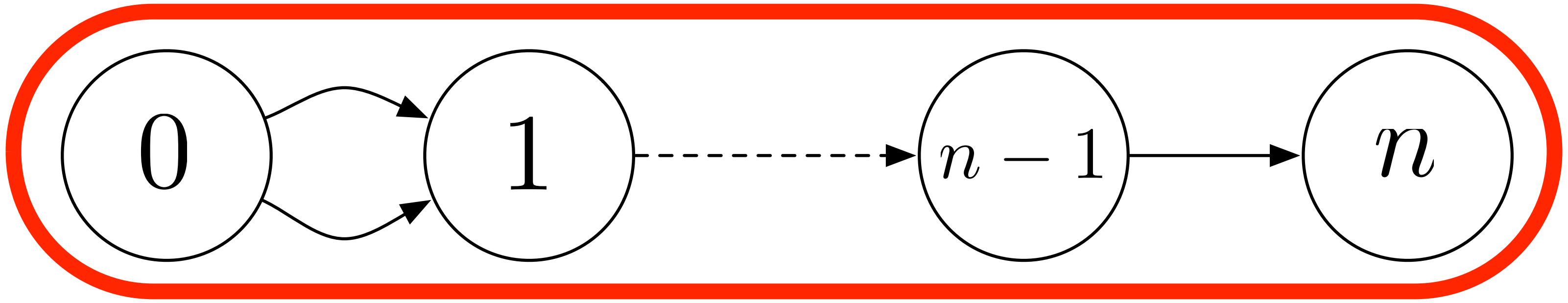
```
int f() {...}
int main()
{
   f();
   f();
}
```

- Spurious cycles degrade both precision and time
  - Spurious information flows
  - Fake cyclic dependence cycle

Easy To Be Large

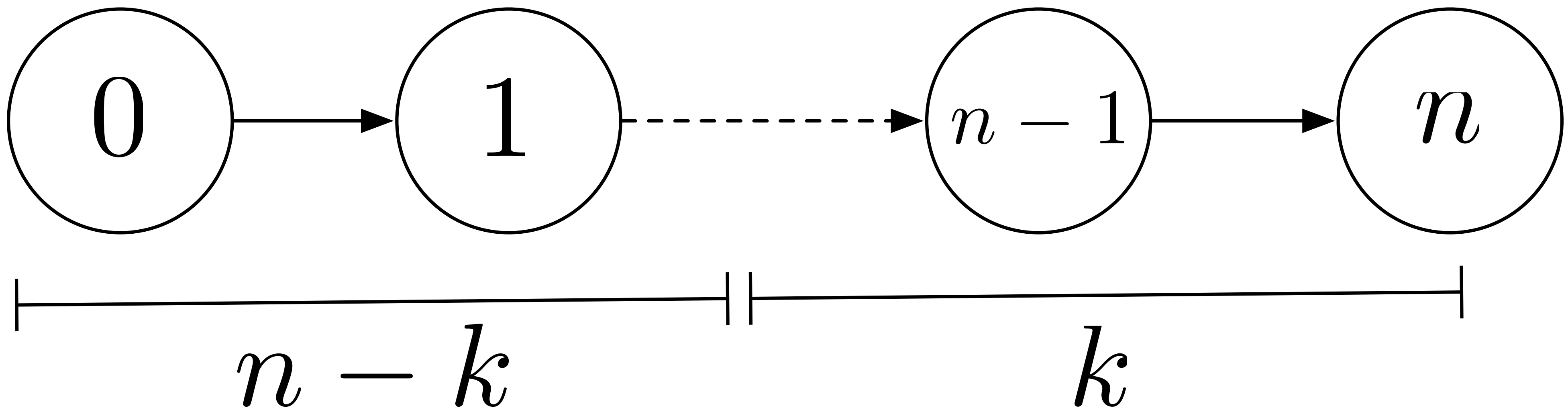


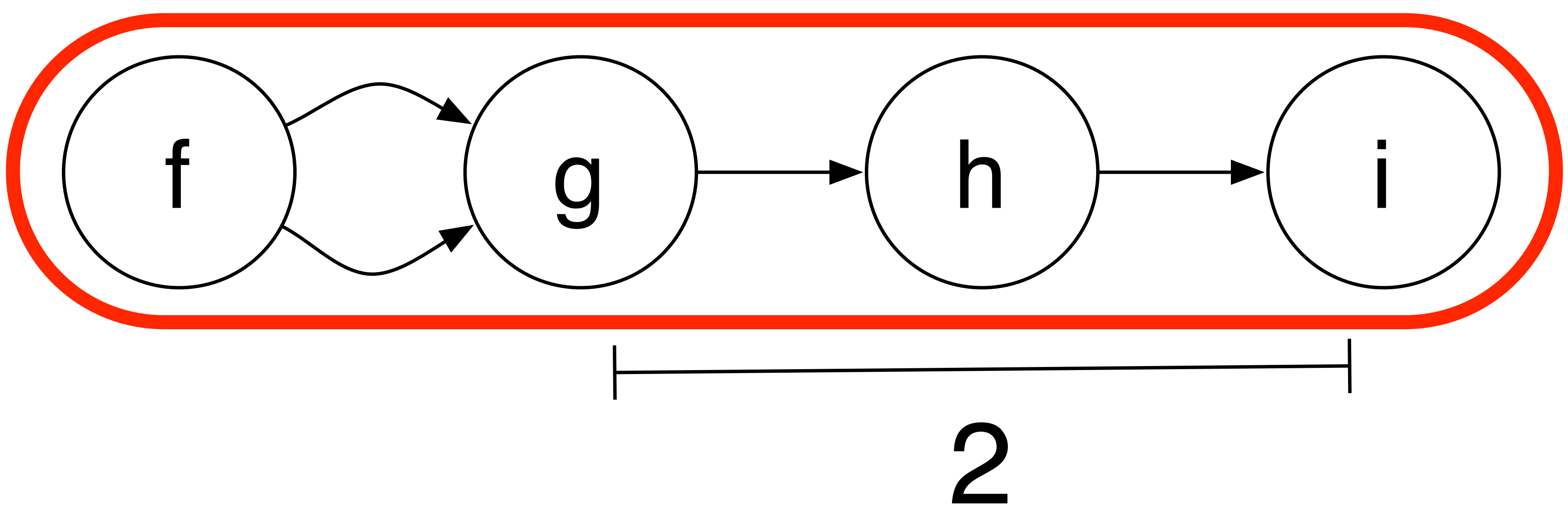
Large Spurious Cycles



Program	Basic-blocks in the largest cycle
spell-1.0	751/782(95%)
gzip-1.2.4a	5,988/6,271(95%)
sed-4.0.8	14,559/14,976(97%)
tar-1.13	10,194/10,800(94%)
wget-1.9	15,249/16,544(92%)
bison-1.875	12,558/18,110(69%)
proftpd-1.3.1	35,386/41,062(86%)
apache-2.2.2	71,719/95,179(75%)

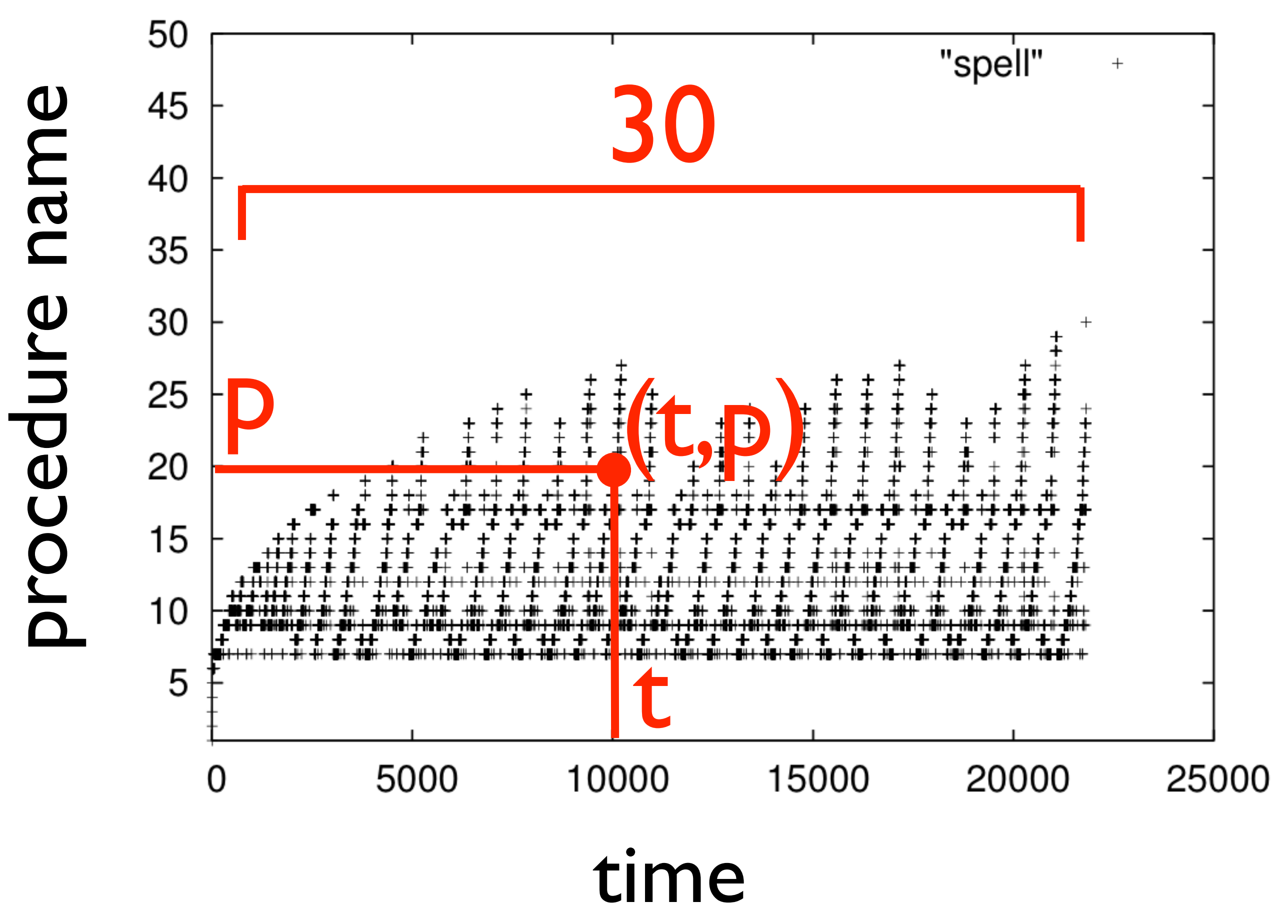
Increasing *k* Does Not Help



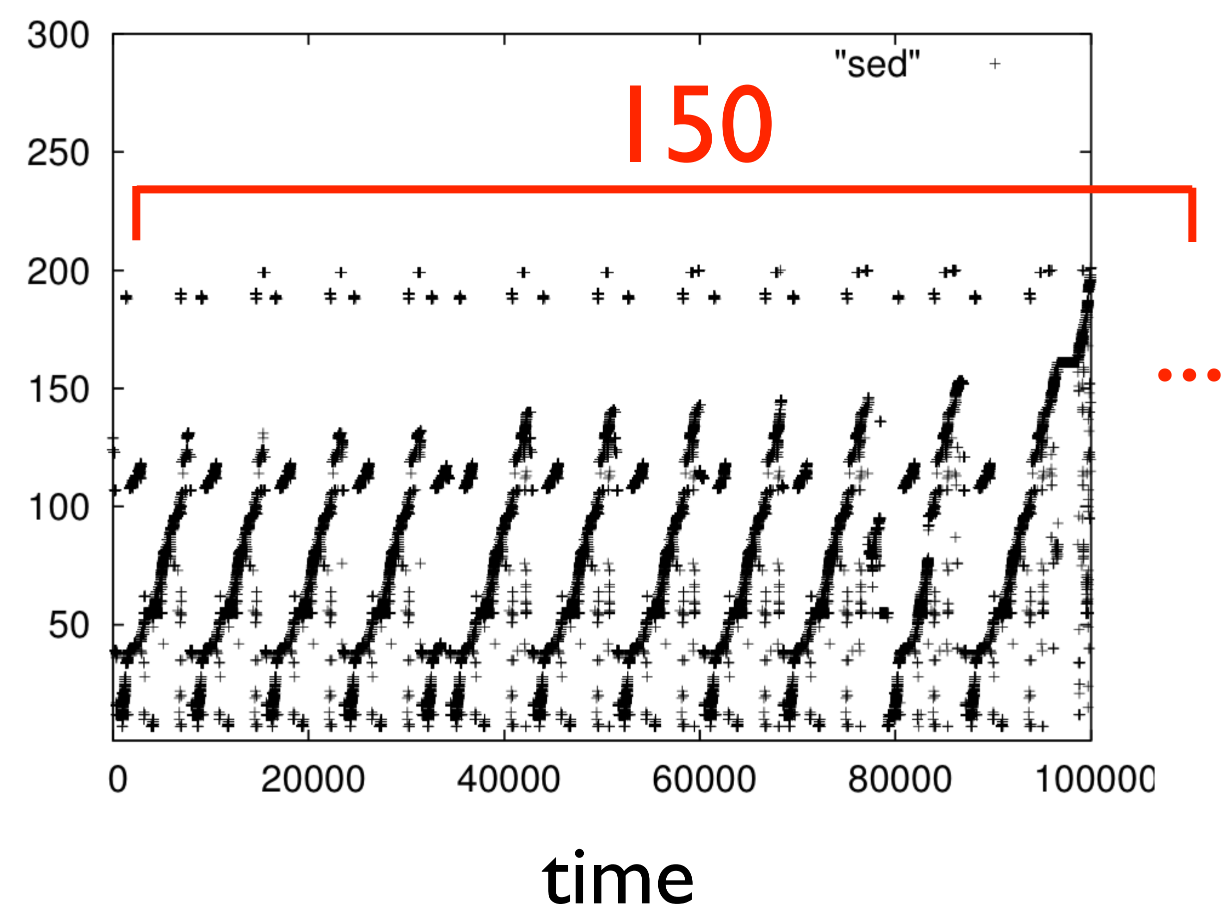


Performance Problems due to Large Spurious Cycle

spell-1.0



sed-4.0.8

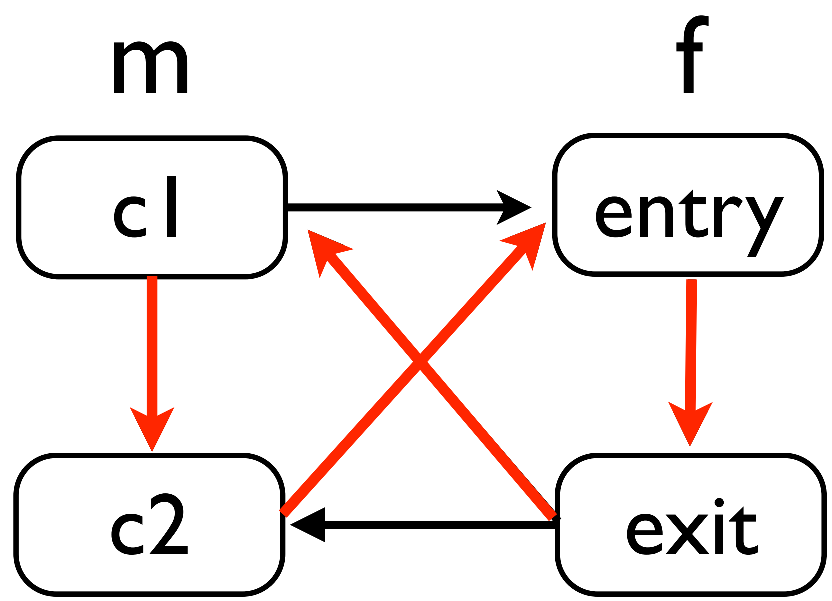


Plan

$$\boxed{\mathsf{Normal}_0 \rightarrow \mathsf{RSS}_0}$$

- In the paper,
  - Generalization for k > 0
  - Details on algorithm, correctness, and precision

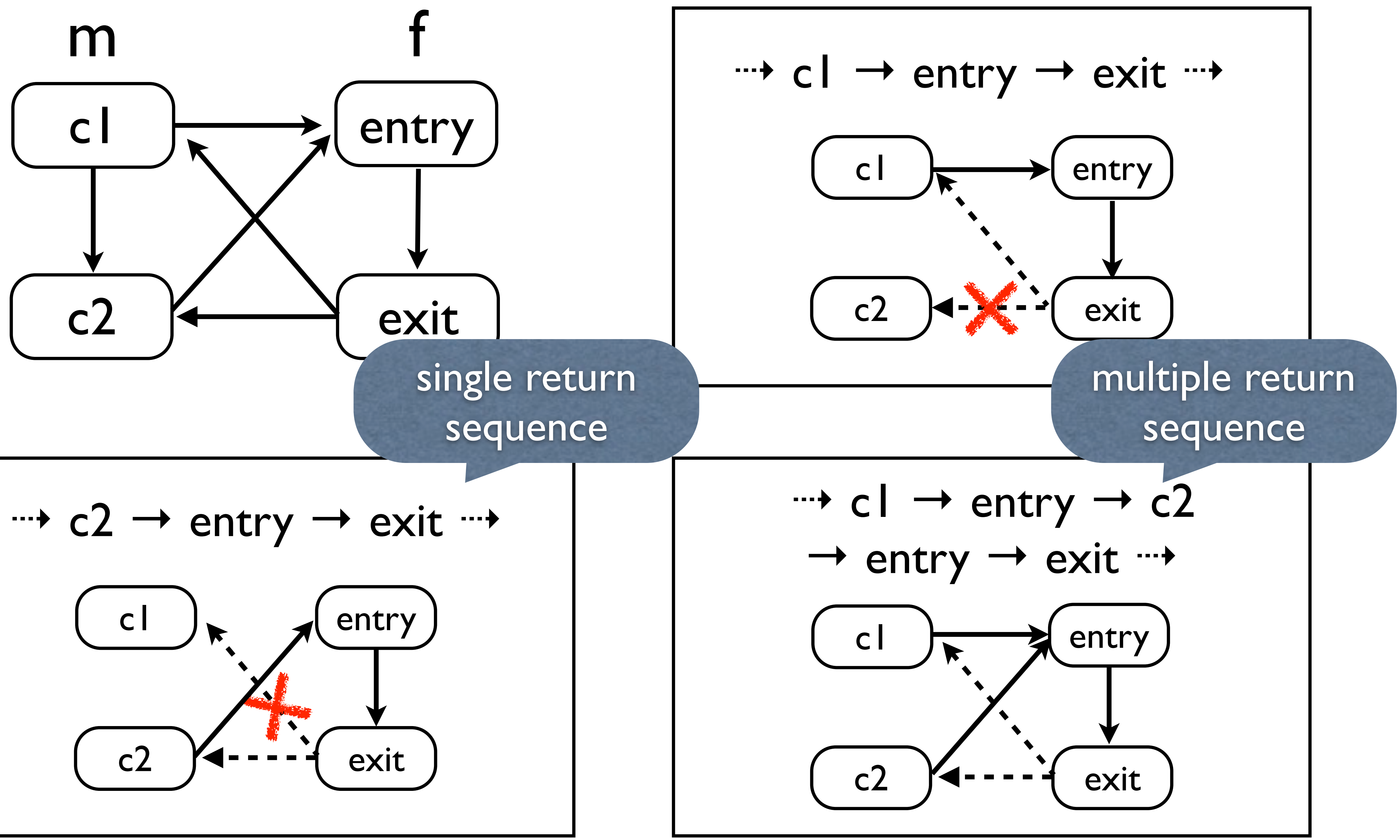
Worklist-based Normal$_0$ Algorithm



⋯→ c2 → entry → exit → c1 → c2 → ...

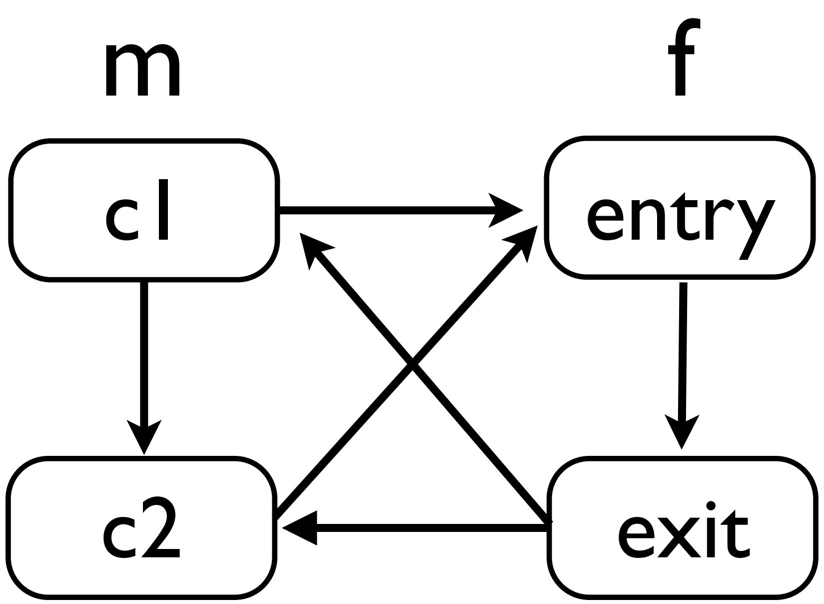
cycle

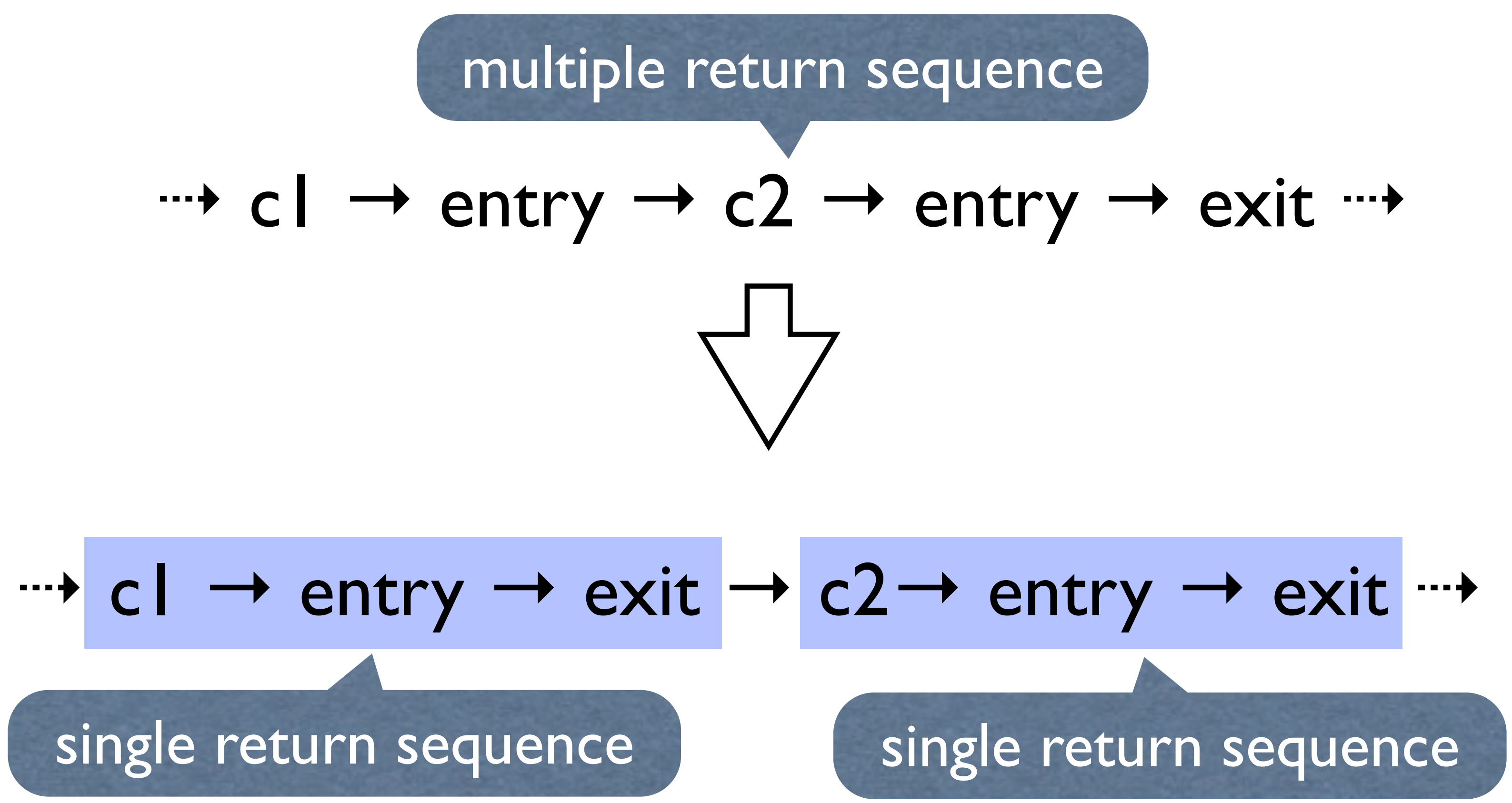
Observation



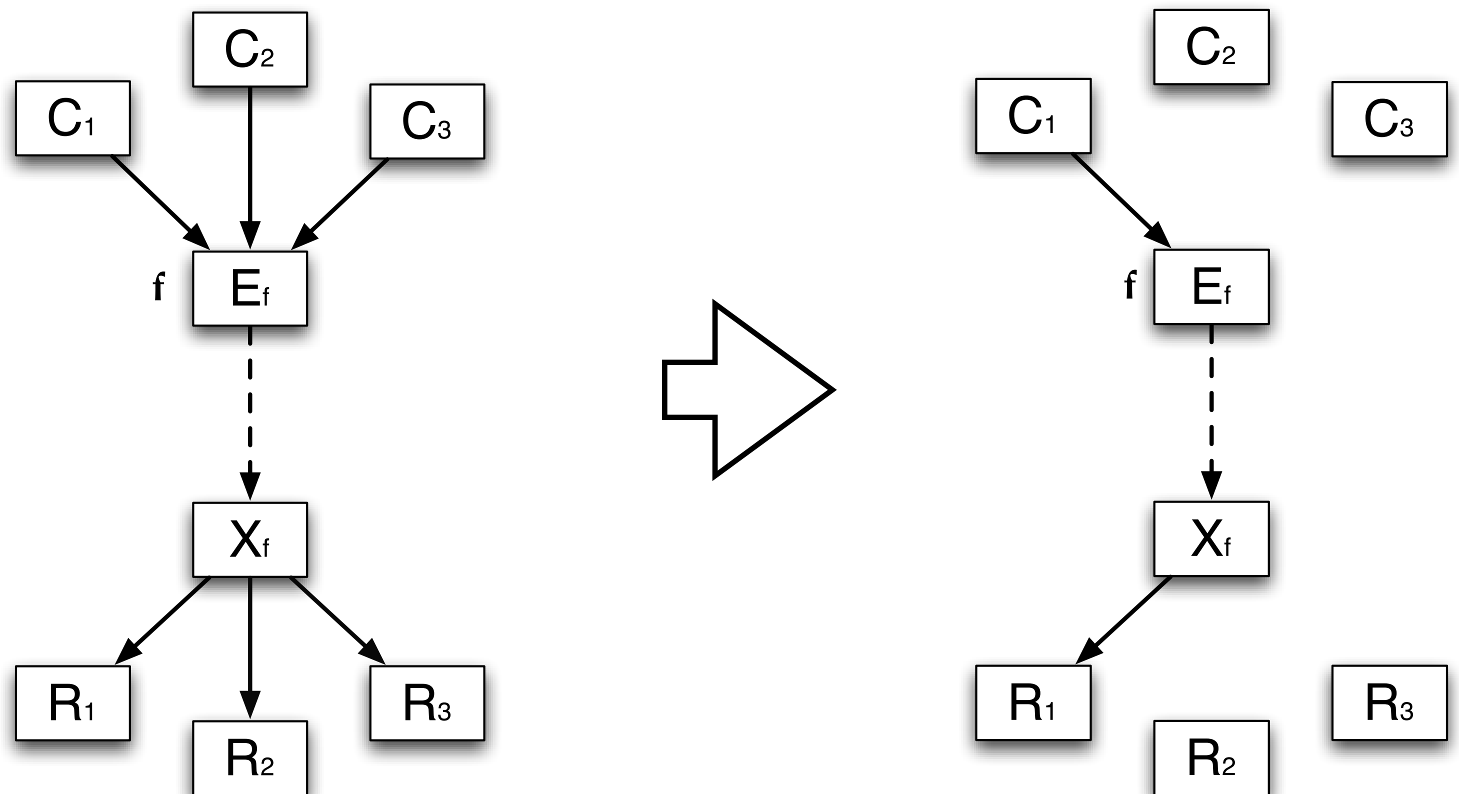
Basic Idea

1. Change M.R.S. into S.R.S
2. Enforce single return to the last called site





One-call-per-procedure



- A procedure is analyzed exclusively for its one particular call-site.
  - Each called procedure is locked.
  - The other call-sites wait until the lock is released.

Controlling Worklist

Prioritize a callee procedure over its call-sites

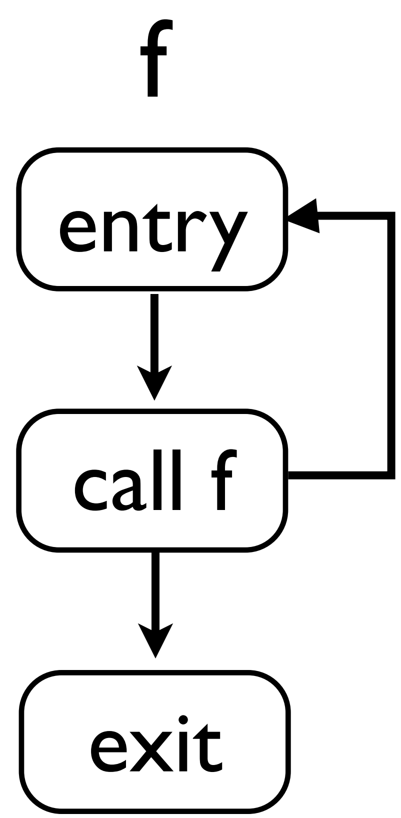
For example,

$W = \{c_2, n_H\}$ → select n_H first

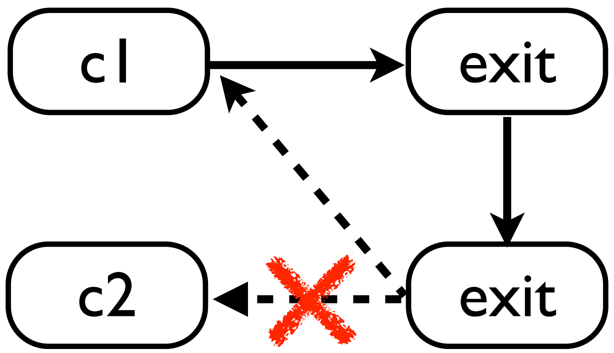
$W = \{c_1, n_H\}$ → select n_H first

Recursion Handling

- Recursive procedures are handled in the same way as the normal algorithm.
- We cannot finish analyzing a recursive procedure without considering other calls in it.



Correctness



- The result is not a fixpoint of the given equation system.
- But still a sound approximation of the program semantics.

Summary

$\delta \in Context = \Delta$
$w \in Work = Node \times \Delta$
$\mathcal{W} \in Worklist = 2^{Work}$
$\mathcal{N} \in Node \times \Delta \to 2^{Node \times \Delta}$
$State = \Delta \to Mem$
$\mathcal{T} \in Table \ = Node \to State$
$\hat{\mathcal{F}} \in Node \to Mem \to Mem$

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

repeat

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

$\quad$ **for all** $(n', \delta') \in \mathcal{N}(n, \delta)$ **do**
$\quad\quad$ **if** $m \not\sqsubseteq \mathcal{T}(n')(\delta')$
$\quad\quad\quad \mathcal{W} := \mathcal{W} \cup \{(n', \delta')\}$
$\quad\quad\quad \mathcal{T}(n')(\delta') := \mathcal{T}(n')(\delta') \sqcup m$
until $\mathcal{W} = \emptyset$

⇒

$\delta \in Context = \Delta$
$w \in Work = Node \times \Delta$
$\mathcal{W} \in Worklist = 2^{Work}$
$\mathcal{N} \in Node \times \Delta \to 2^{Node \times \Delta}$
$State = \Delta \to Mem$
$\mathcal{T} \in Table \ = Node \to State$
$\hat{\mathcal{F}} \in Node \to Mem \to Mem$
$ReturnSite \in ProcName \to Work$

$FixpointIterate \ (\mathcal{W}, \mathcal{T}) =$
$ReturnSite := \emptyset$
repeat
$\quad \mathcal{S} := \{(call^{g,\_}_{\_}, \_) \in \mathcal{W} \mid (n_h, \_) \in \mathcal{W} \land \mathsf{reach}(g, h) \land \neg\mathsf{recursive}(g)\}$
$\quad (n, \delta) := \mathsf{choose}(\ \mathcal{W} \setminus \mathcal{S}\ )$
$\quad m := \hat{\mathcal{F}} \ n \ (\mathcal{T}(n)(\delta))$
$\quad$ **if** $n = call^{g,r}_{f} \land \neg\mathsf{recursive}(g)$ **then**
$\quad\quad ReturnSite(g) := (r, \delta)$
$\quad$ **if** $n = exit_g \land \neg\mathsf{recursive}(g)$ **then**
$\quad\quad (r, \delta_r) := ReturnSite(g)$
$\quad\quad$ **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** $(n', \delta') \in \mathcal{N}(n$ …
$\quad\quad\quad$ **if** $m \not\sqsubseteq \mathcal{T}($ …
$\quad\quad\quad\quad \mathcal{W} :=$ …
$\quad\quad\quad\quad \mathcal{T}(n')($ …
until $\mathcal{W} = \emptyset$

Control worklist

Recursion handling

Enforce single return

Experiments

- Sparrow (The Early Bird) is an industrialized static analyzer
  - interval-domain-based abstract interpreter
- 11 open-source software packages

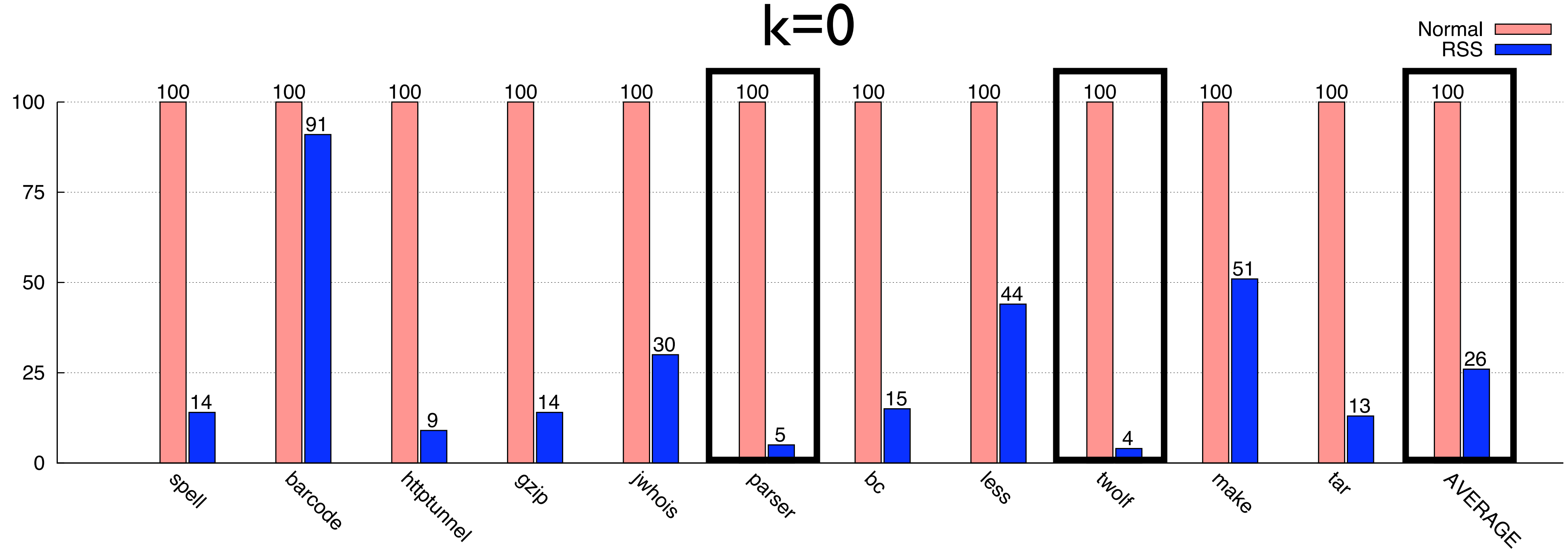
Normal$_k$ vs. RSS$_k$

Normal$_k$ vs. RSS$_{k+1}$

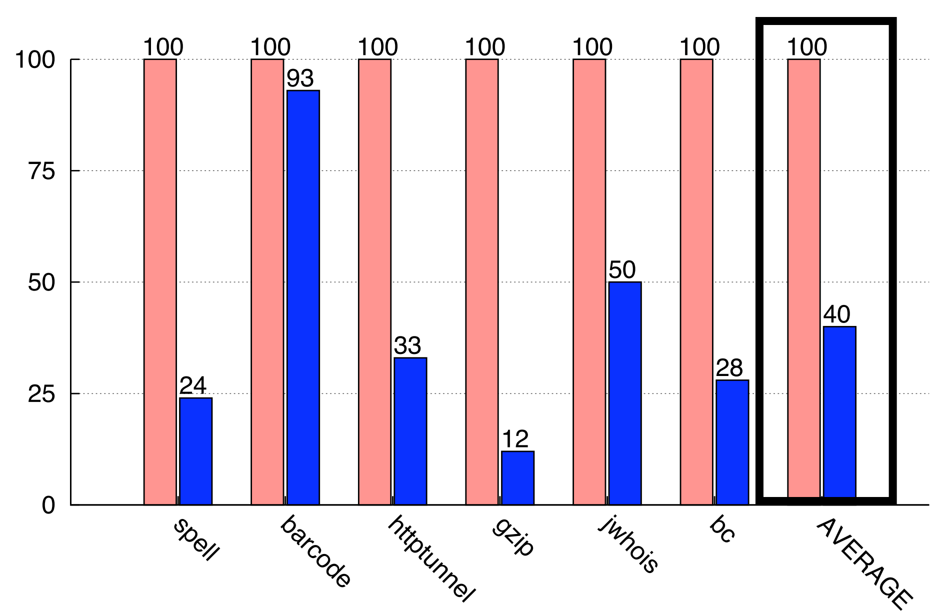
Program	LOC	#Basic-Blocks
spell-1.0	2,213	782
barcode-0.96	4,460	2,634
httptunnel-3.3	6,174	2,757
gzip-1.2.4a	7,327	6,271
jwhois-3.0.1	9,344	5,147
parser	10,900	9,298
bc-1.06	13,093	4,924
less-290	18,449	7,754
twolf	19,700	14,610
tar-1.13	20,258	10,800
make-3.76.1	27,304	11,061

Normal$_k$ vs. RSS$_k$

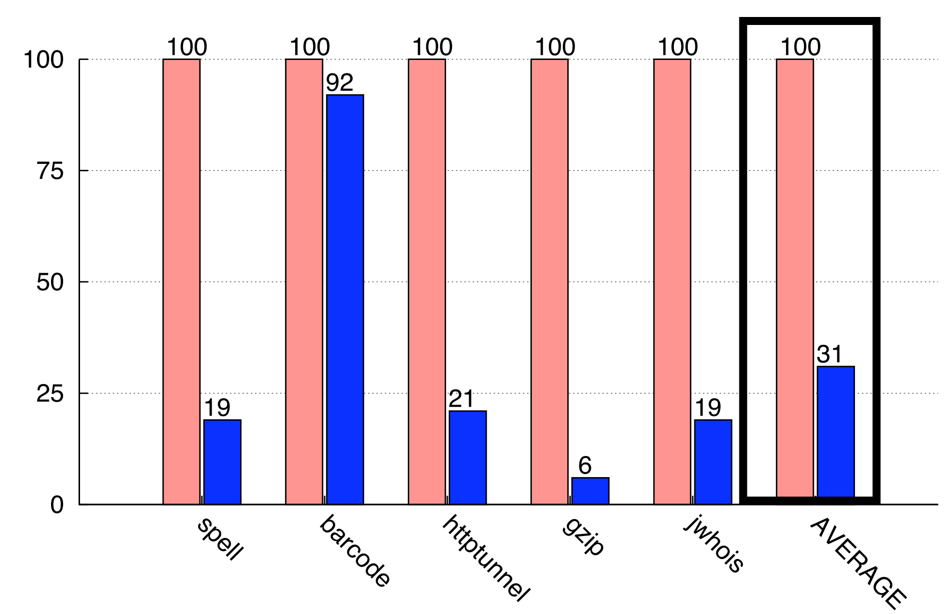
k=0



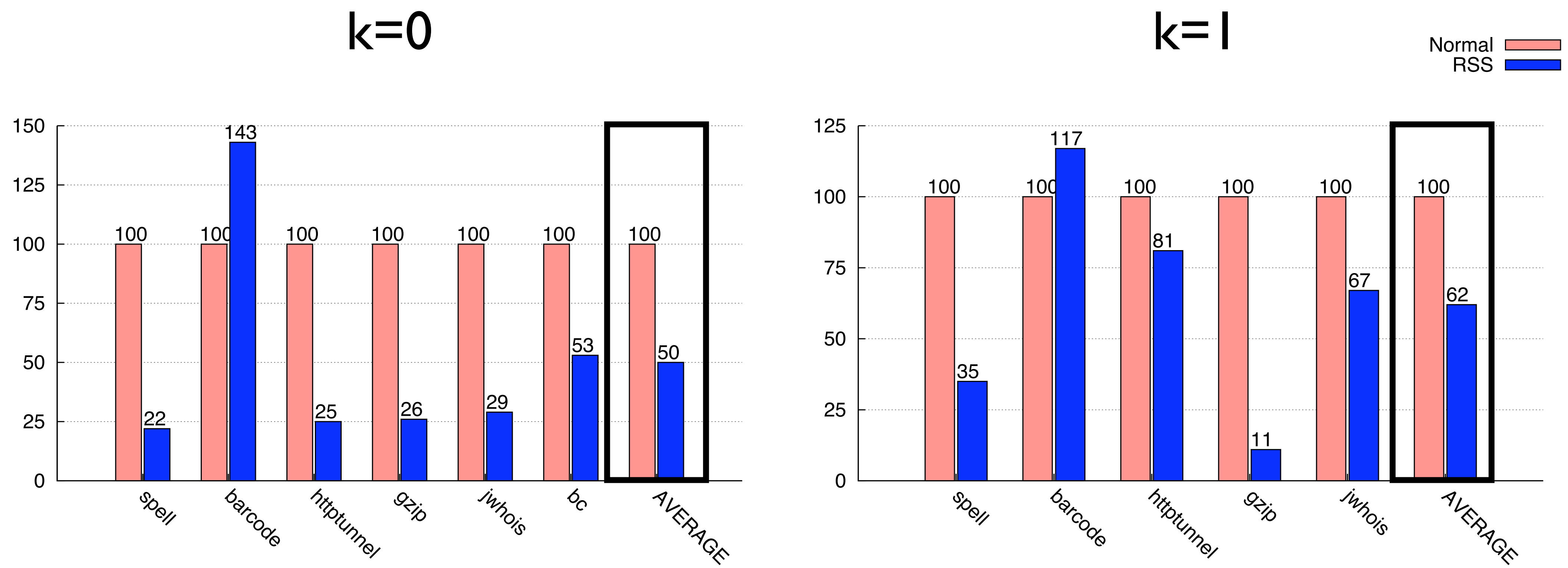
k=1



k=2

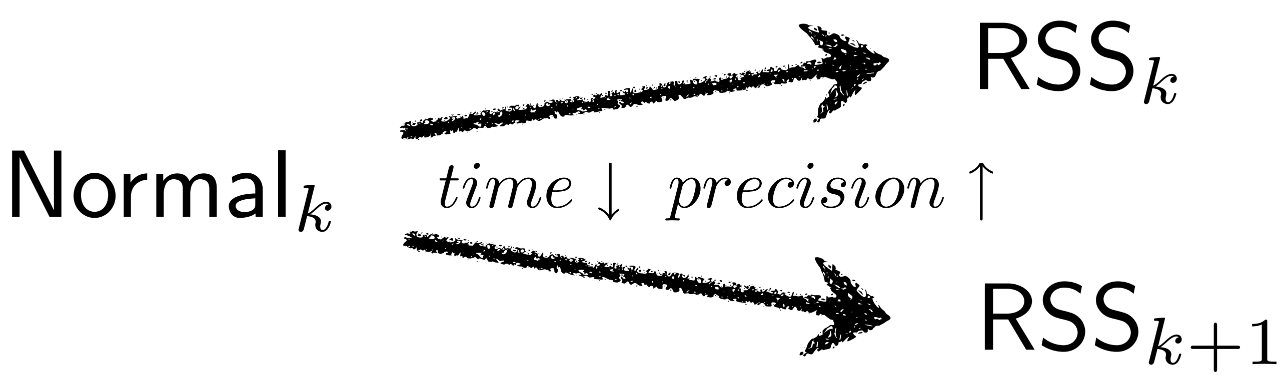


Normal$_k$ vs. RSS$_{k+1}$



Conclusion

- One key reason why less accurate context-sensitivity makes the analysis very slow.
- A simple algorithm that mitigates the problem.



Thank you