Static Slicing of Concurrent Programs

- An Evaluation -

Dennis Giffhorn, Christian Hammer

Universität Passau

October 11, 2007

Initial situation

Precise slicing of concurrent programs with procedures and recursion

- J. Krinke: Context-Sensitive Slicing of Concurrent Programs (ESEC/FSE, September 2003)
- M. Nanda and S. Ramesh: Interprocedural slicing of multithreaded programs with applications to Java (ACM TOPLAS, 2006)

Extending SDGs for concurrent programs

Interference dependence

- Statement **n** ist interference dependent on statement **m**, if:
  1. **n** uses variable v and **m** defines v
  2. **m** and **n** are executed concurrently
- No execution order between **n** and **m**

→ Interference dependence is not transitive

Extending SDGs for concurrent programs

Interference dependence

- Statement **n** ist interference dependent on statement **m**, if:
  1. **n** uses variable v and **m** defines v
  2. **m** and **n** are executed concurrently
- No execution order between **n** and **m**

→ Interference dependence is not transitive

Extending SDGs for concurrent programs

Interference dependence

- Statement **n** ist interference dependent on statement **m**, if:
  1. **n** uses variable v and **m** defines v
  2. **m** and **n** are executed concurrently
- No execution order between **n** and **m**

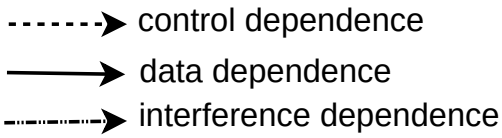
$\rightarrow$ Interference dependence is not transitive

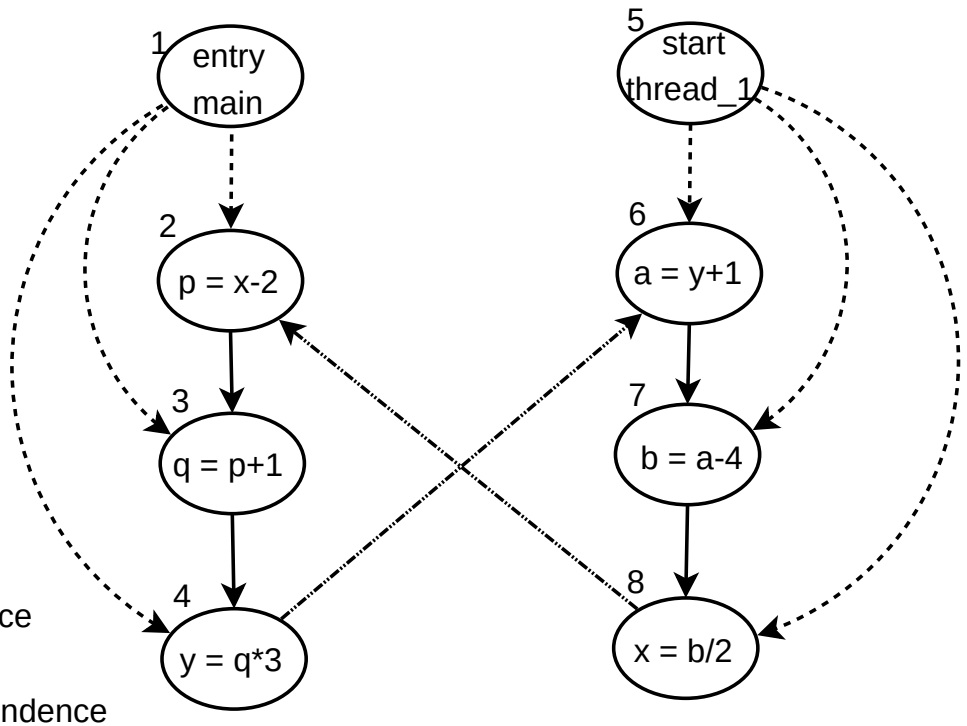
Example – slice for node 7

```
int x = 0, y = 1;

1: main ()
2:    int p = x - 2;
3:    int q = p + 1;
4:    y = q * 3;

5: thread_1 ()
6:    int a = y + 1;
7:    int b = a - 4;
8:    x = b / 2;
```



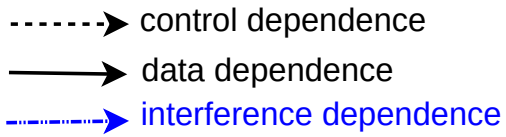


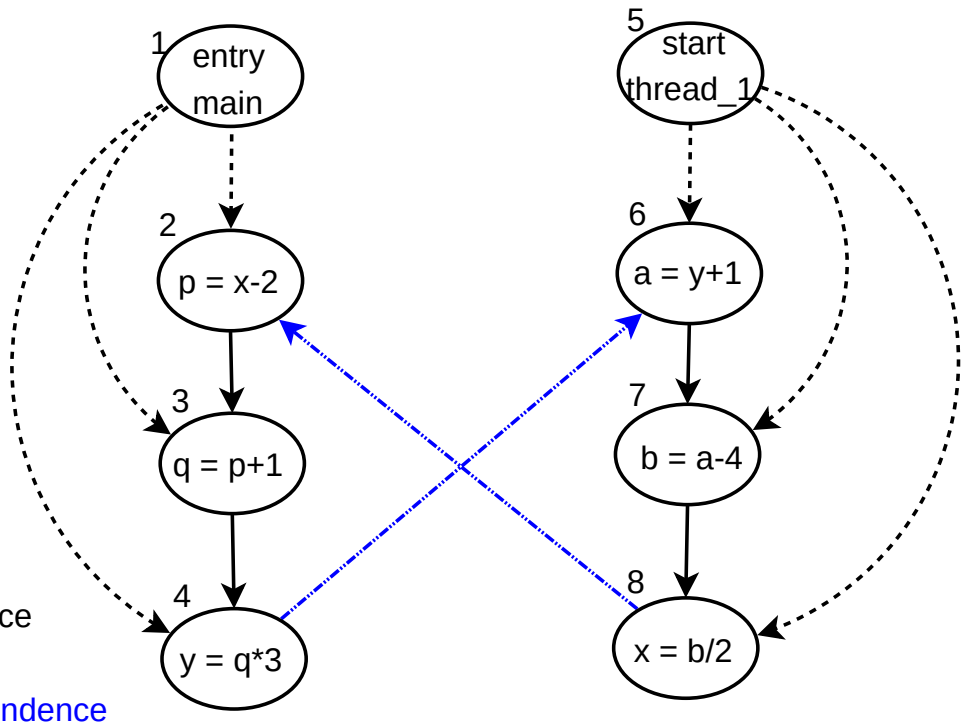
Example – slice for node 7

```
int x = 0, y = 1;

1: main ()
2:    int p = x - 2;
3:    int q = p + 1;
4:    y = q * 3;

5: thread_1 ()
6:    int a = y + 1;
7:    int b = a - 4;
8:    x = b / 2;
```



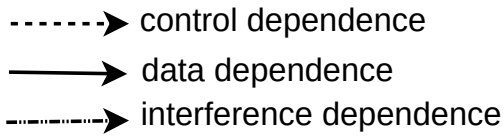


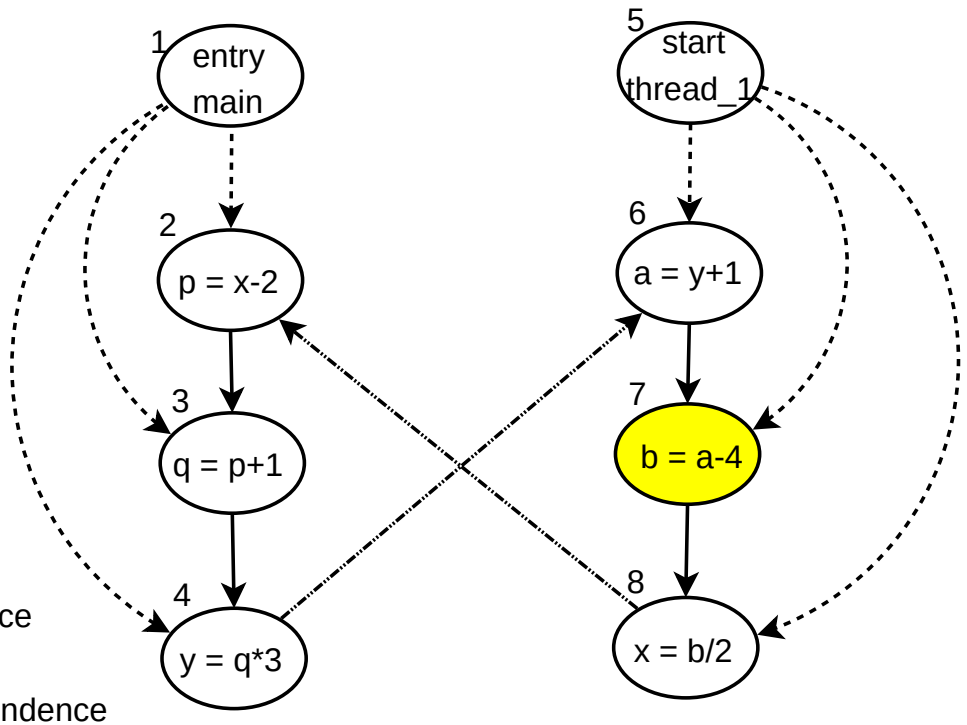
Example – slice for node 7

```
int x = 0, y = 1;

1: main ()
2:    int p = x - 2;
3:    int q = p + 1;
4:    y = q * 3;

5: thread_1 ()
6:    int a = y + 1;
7:    int b = a - 4;
8:    x = b / 2;
```



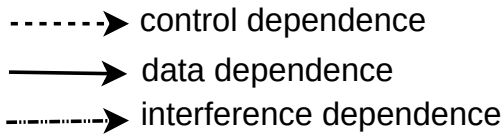


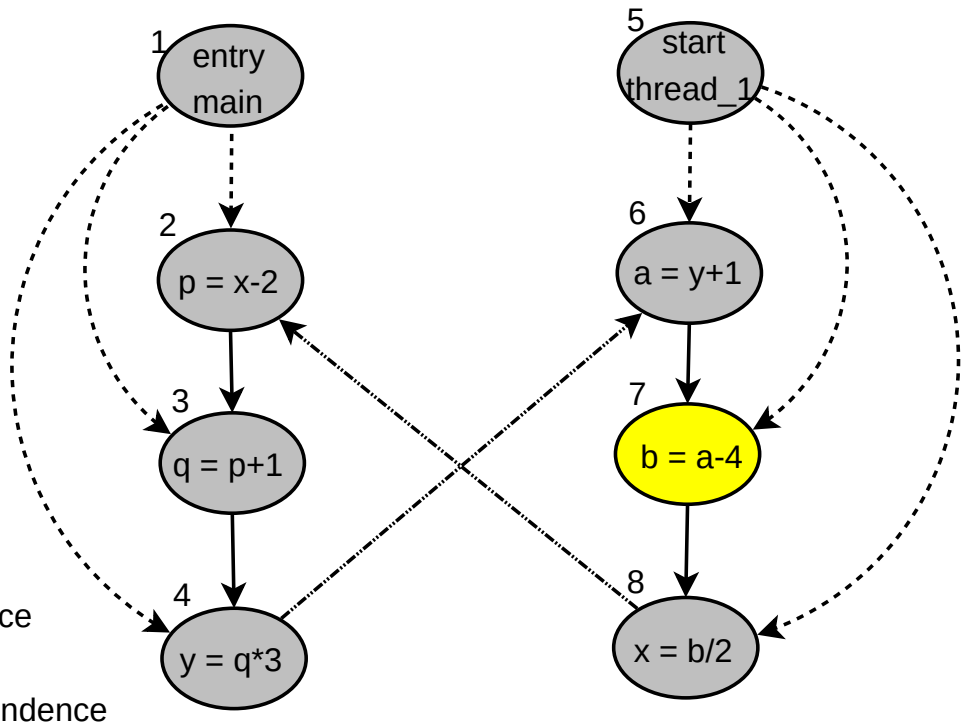
Example – slice for node 7

```
int x = 0, y = 1;

1: main ()
2:    int p = x - 2;
3:    int q = p + 1;
4:    y = q * 3;

5: thread_1 ()
6:    int a = y + 1;
7:    int b = a - 4;
8:    x = b / 2;
```



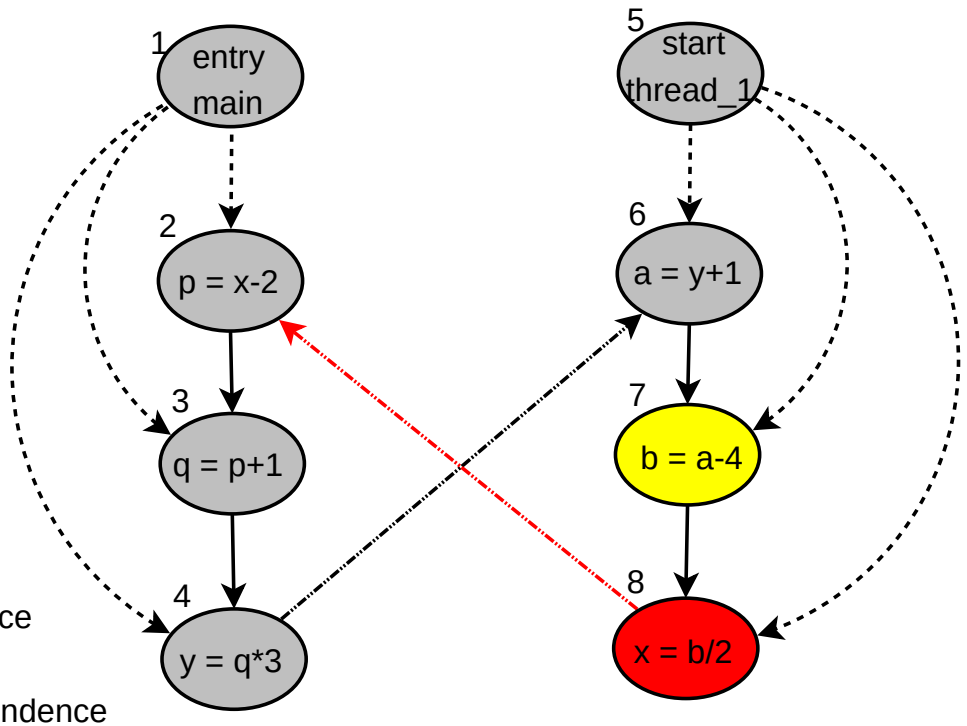


Example – imprecise result

```
int x = 0, y = 1;

1: main ()
2:    int p = x - 2;
3:    int q = p + 1;
4:    y = q * 3;

5: thread_1 ()
6:    int a = y + 1;
7:    int b = a - 4;
8:    x = b / 2;
```



- Node 8 cannot influence node 7
  - $\Rightarrow$ Time travel

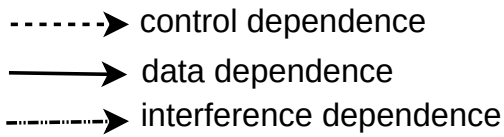
$\rightarrow$ Solution: remember where threads are left

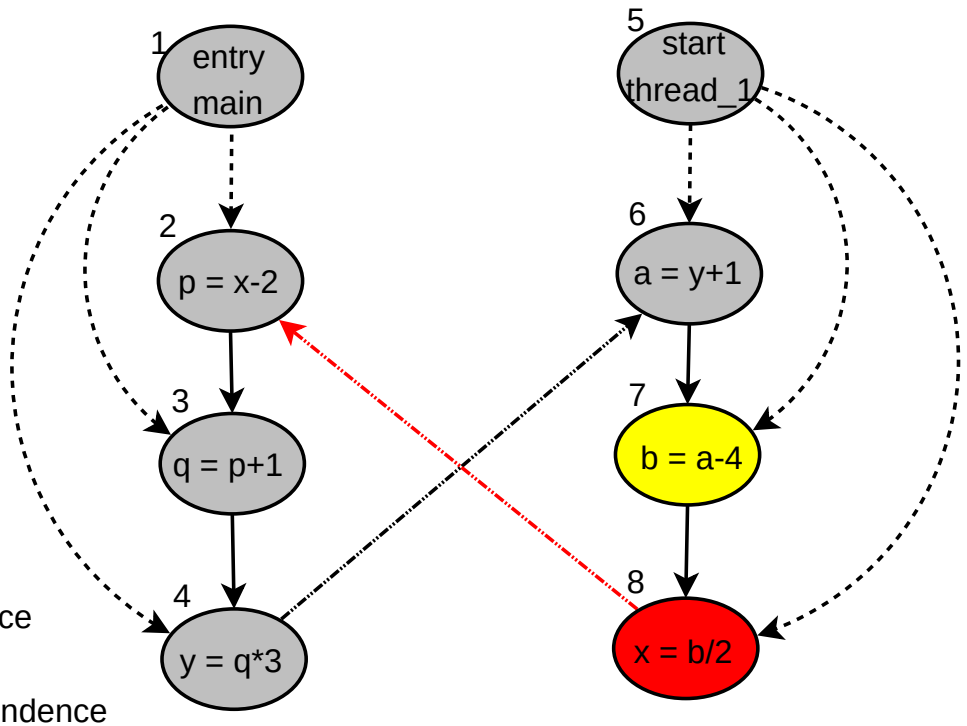
Example – imprecise result

```
int x = 0, y = 1;

1: main ()
2:    int p = x - 2;
3:    int q = p + 1;
4:    y = q * 3;

5: thread_1 ()
6:    int a = y + 1;
7:    int b = a - 4;
8:    x = b / 2;
```





- Node 8 cannot influence node 7
  - $\Rightarrow$ Time travel

$\rightarrow$ Solution: remember where threads are left

Problem of this approach

- Nodes can be visited multiple times
  - Remember where each thread was left to reach a node
  - → Nodes are annotated with *thread states*
  - Visited as often as its thread state annotations differ
  - → Worst case number of visits: $O(|nodes|^{|threads|})$

Problem of this approach

- Nodes can be visited multiple times
  - Remember where each thread was left to reach a node

  → Nodes are annotated with *thread states*

  - Visited as often as its thread state annotations differ

  → Worst case number of visits: $O(|nodes|^{|threads|})$

Problem of this approach

- Nodes can be visited multiple times
  - Remember where each thread was left to reach a node

  → Nodes are annotated with *thread states*

  - Visited as often as its thread state annotations differ

  → Worst case number of visits: $O(|nodes|^{|threads|})$

Problem of this approach

- Nodes can be visited multiple times
  - Remember where each thread was left to reach a node

  → Nodes are annotated with *thread states*

  - Visited as often as its thread state annotations differ

  → Worst case number of visits: $O(|nodes|^{|threads|})$

Problem of this approach

- Nodes can be visited multiple times
  - Remember where each thread was left to reach a node

  → Nodes are annotated with *thread states*

  - Visited as often as its thread state annotations differ

  → Worst case number of visits: $O(|nodes|^{|threads|})$

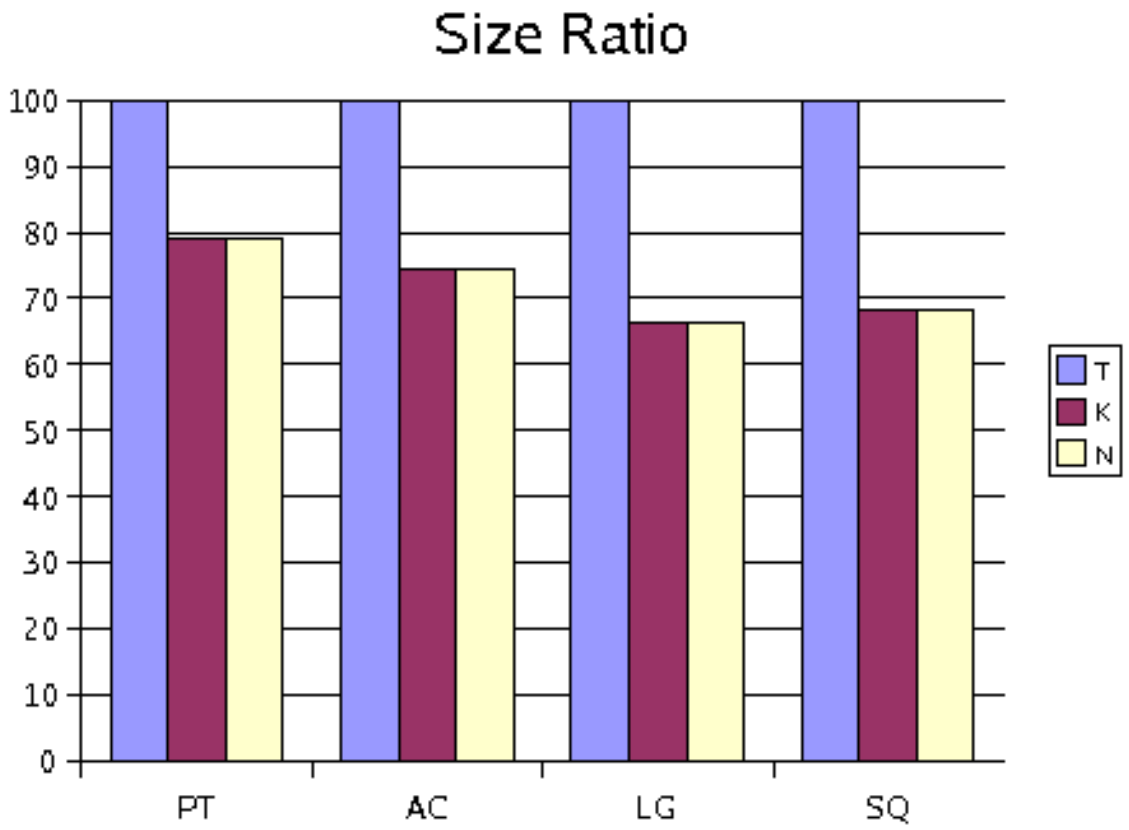
Sample case study

- Algorithms
  - T – transitive approximation
  - K – optimized version of Krinke's algorithm
  - N – optimized version of Nanda's algorithm

- 4 sample programs

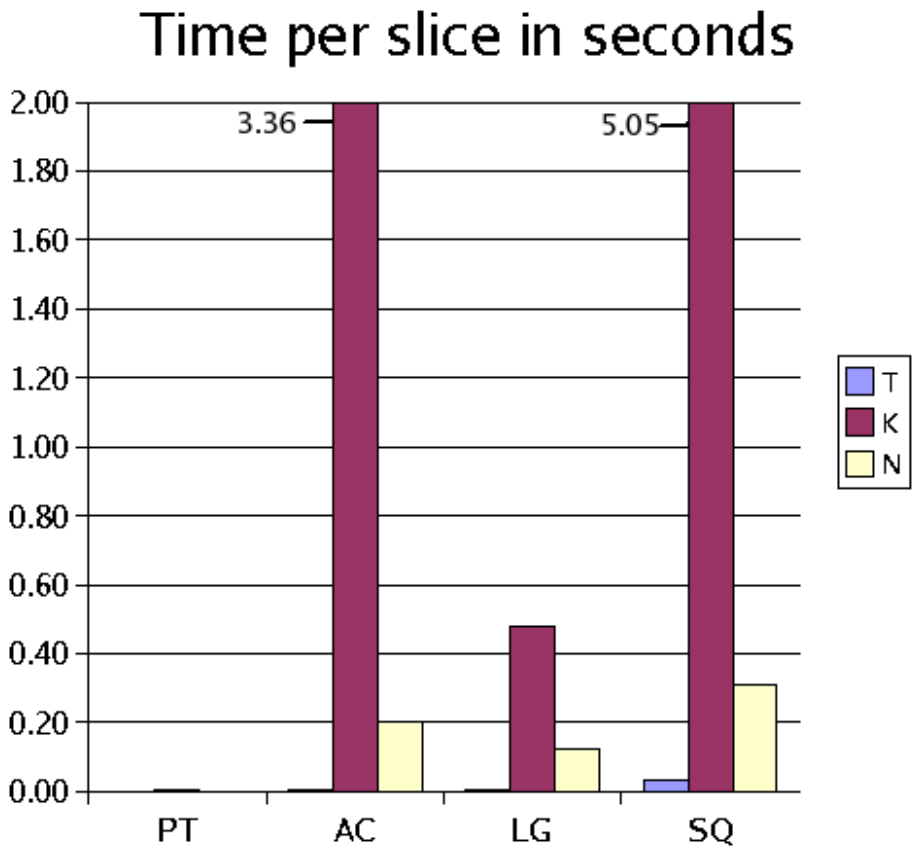
Name	Nodes	Edges	Classes	Methods	Threads
PrecisionTest	328	904	6	10	2
AlarmClock	4085	13842	17	74	2
LaplaceGrid	10022	100730	22	95	3
SharedQueue	17998	139480	23	122	3

Average size



- Gain of precision: up to 35%

Average speed



- K and N are much more expensive than T

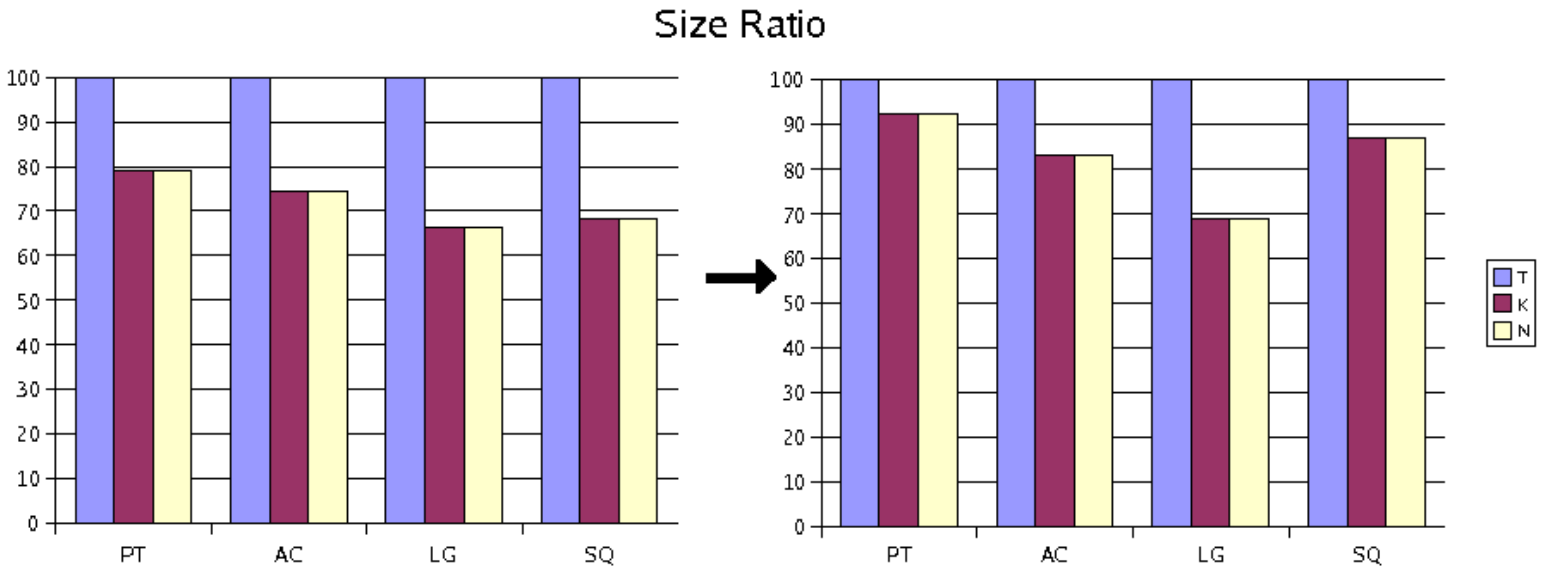
Stress test

- Now we double the number of threads in these programs
- What will happen?

Stress test

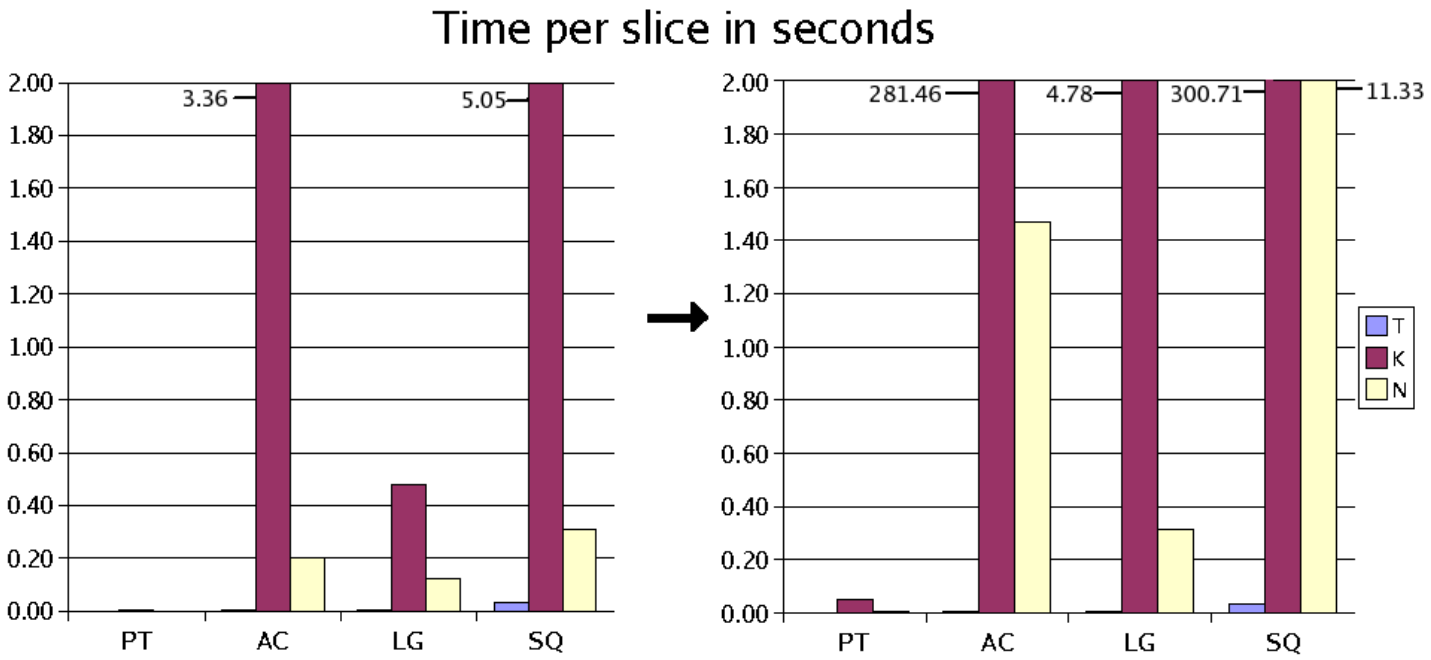
- Now we double the number of threads in these programs
- What will happen?

Average size



- The gain of precision decreases...

Average speed



- ...and the costs explode

Conclusion

⇒ **Precise slicing of concurrent programs does not scale for real-world programs**

⇒ **We need a trade-off between precision and speed**

- K-limitation for time travel detection

Conclusion

⇒ **Precise slicing of concurrent programs does not scale for real-world programs**

⇒ **We need a trade-off between precision and speed**

- K-limitation for time travel detection

Questions?