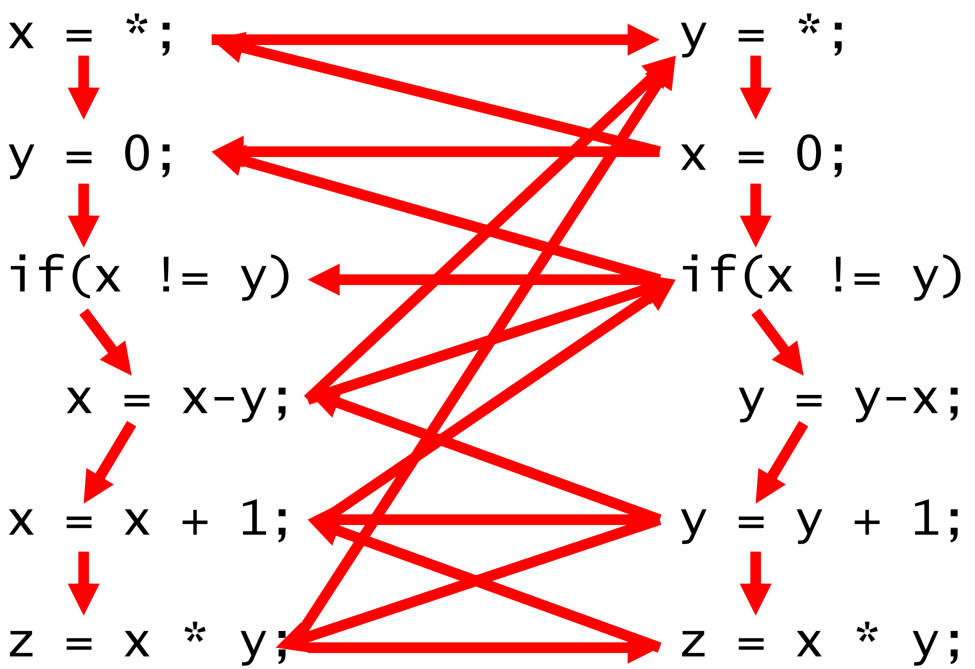
Approccio SWARM:
tool VeriSmart

Salvatore La Torre

Dipartimento di Informatica
Università degli Studi di Salerno

Concurrency makes bug finding harder.

State space explosion (i.e., large number of interleavings):



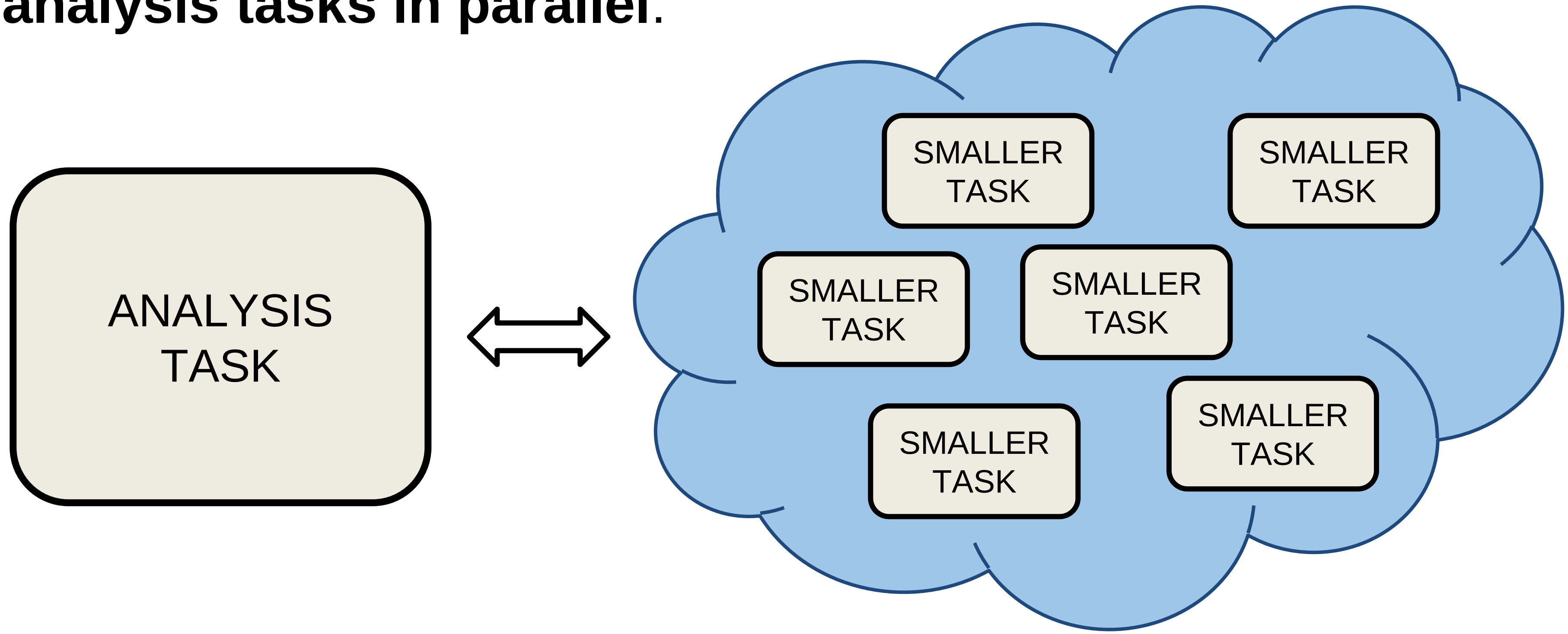
(… and many more)

Problem: modern hardware means concurrency is everywhere

$\Rightarrow$ software is increasingly concurrent

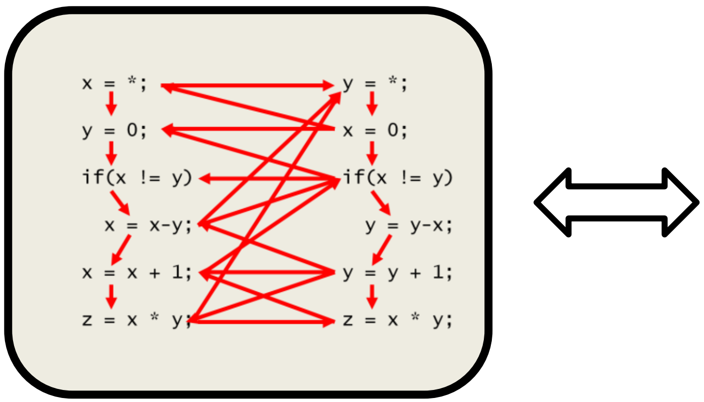
Concurrency makes bug finding *easier*.

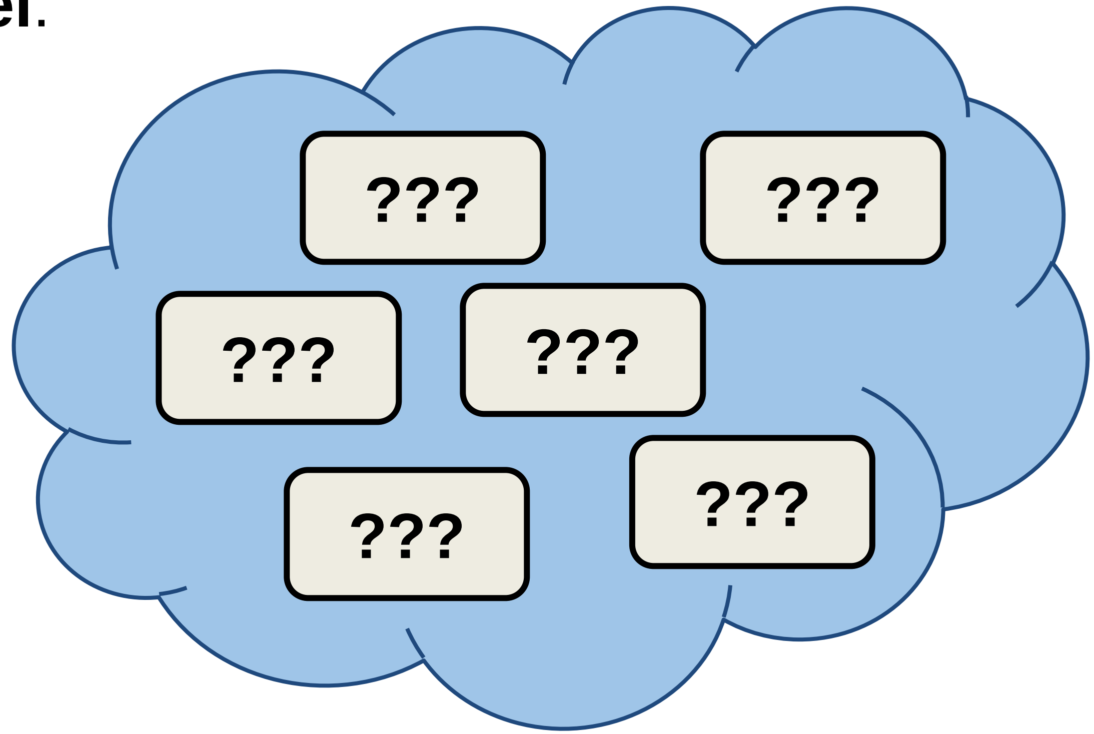
Concurrent hardware allows us to **run many (smaller) analysis tasks in parallel**:



Concurrency makes bug finding *easier*.

Concurrent hardware allows us to **run many (smaller) analysis tasks in parallel**:





How can we partition a task into **independent smaller** tasks?

Strategy competition vs. task competition

Strategy competition: run **different settings** on **same task**

(**first counterexample** “wins” and aborts other tasks)

Swarm Verification Techniques

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce

> “anticipate the appearance of systems with large numbers of CPU cores, but without matching increases in clockspeeds… describe a model checking strategy that leverages this trend”

Abs ... in
the ... so
ben ... se
in processing speeds, ... started redirecting their efforts to the development of multi-core systems. For the near-term future, we can anticipate the appearance of systems with large numbers of CPU cores, but without matching increases in clockspeeds. We will describe a model checking strategy that can allow us to leverage this trend, and that allows us to tackle significantly larger problem sizes than before.

Index Terms—software engineering tools and techniques, logic model checking, distributed algorithms, software verification.

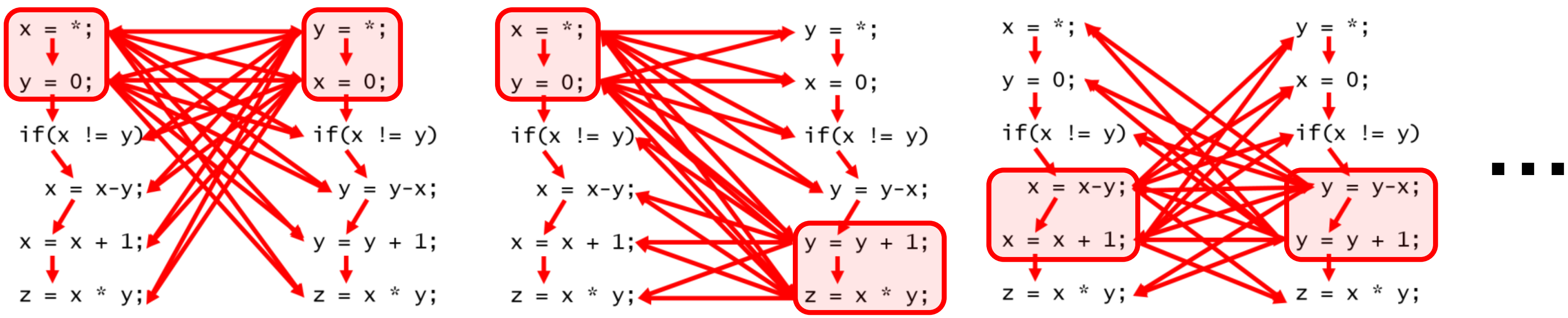
Task competition: run **same prover** (setting) on **different tasks**

$\Rightarrow$ How can we partition a task into **independent smaller** tasks?

Reduced interleaving instances

Our goal:

Split set of **interleavings** $\mathcal{I}_k(P)$ into **subsets** that can be **analyzed symbolically** and **independently**.



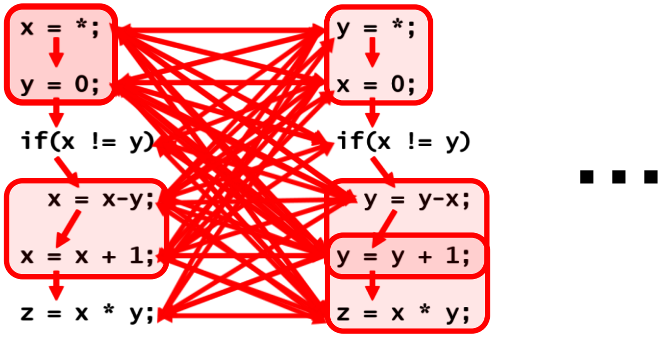
Our solution:

Derive program **variants** P_ϑ that **allow context switches only** in **subsets** of statements (**tiles**) s.t. $\mathcal{I}_k(P) = \bigcup_\vartheta\ \mathcal{I}_k(P_\vartheta)$.

Reduced interleaving instances

Goal:

Split set of **interleavings** $\mathcal{I}_k(P)$ into **subsets** that can be **analyzed symbolically** and **independently**.



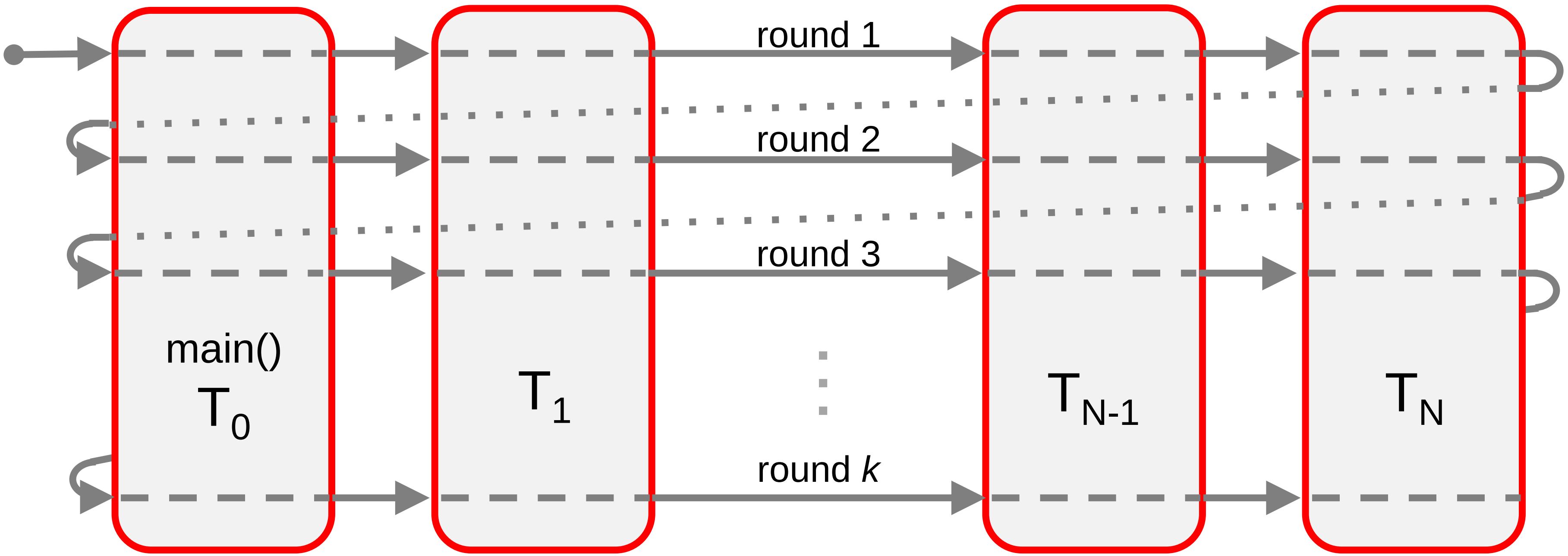
Idea:

Derive program **variants** P_ϑ that **allow context switches only** in **subsets** of statements (**tiles**) s.t. $\mathcal{I}_k(P) = \bigcup_\vartheta \mathcal{I}_k(P_\vartheta)$.

Tiling Threads

Tiling threads

Assumption: bounded concurrent programs



- finite #threads, fixed (but arbitrary) schedule
  - captures all bounded round-robin computations for given bound
- bugs manifest within very few rounds [Musuvathi, Qadeer, PLDI'07]

Tiling threads

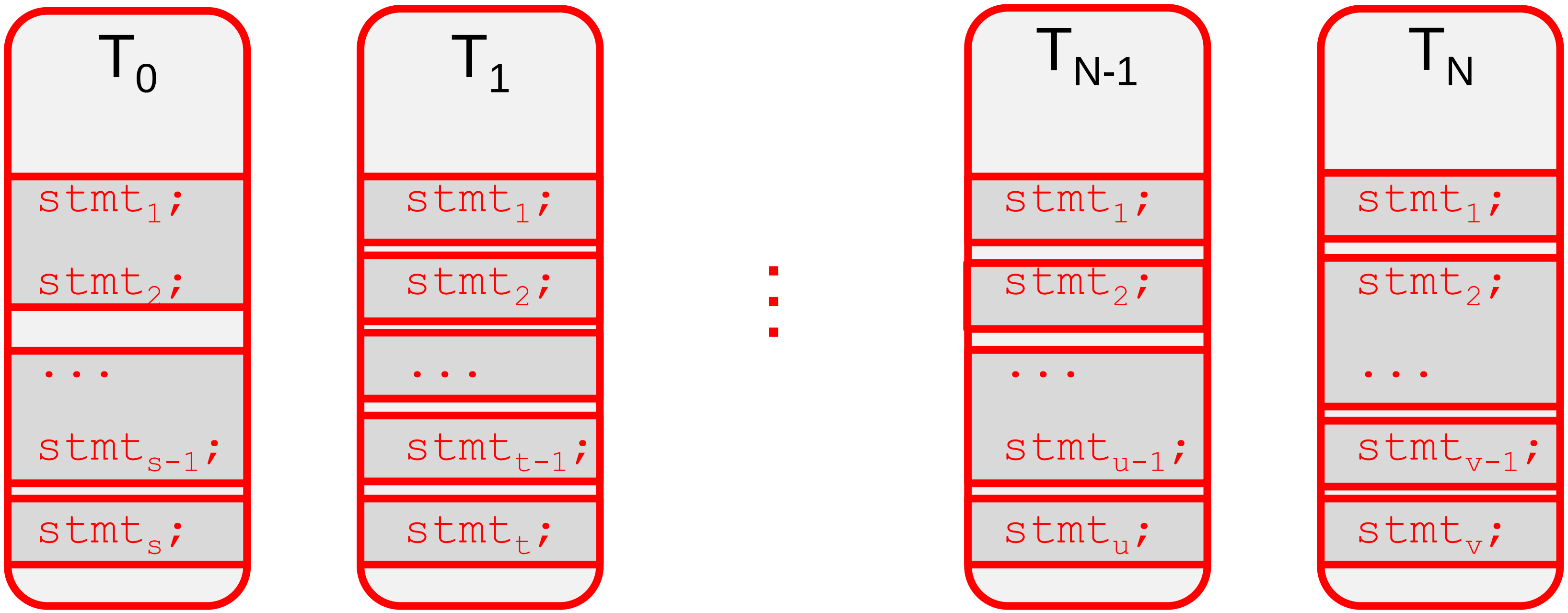
Assumption: bounded concurrent programs

T_0	T_1		T_{N-1}	T_N
$\texttt{stmt}_1$;	$\texttt{stmt}_1$;		$\texttt{stmt}_1$;	$\texttt{stmt}_1$;
$\texttt{stmt}_2$;	$\texttt{stmt}_2$;	⋮	$\texttt{stmt}_2$;	$\texttt{stmt}_2$;
...	...		...	...
$\texttt{stmt}_{s-1}$;	$\texttt{stmt}_{t-1}$;		$\texttt{stmt}_{u-1}$;	$\texttt{stmt}_{v-1}$;
$\texttt{stmt}_s$;	$\texttt{stmt}_t$;		$\texttt{stmt}_u$;	$\texttt{stmt}_v$;

- finite #stmts
- control can only go forward
  - simplifies analysis and tiling

Tiling threads

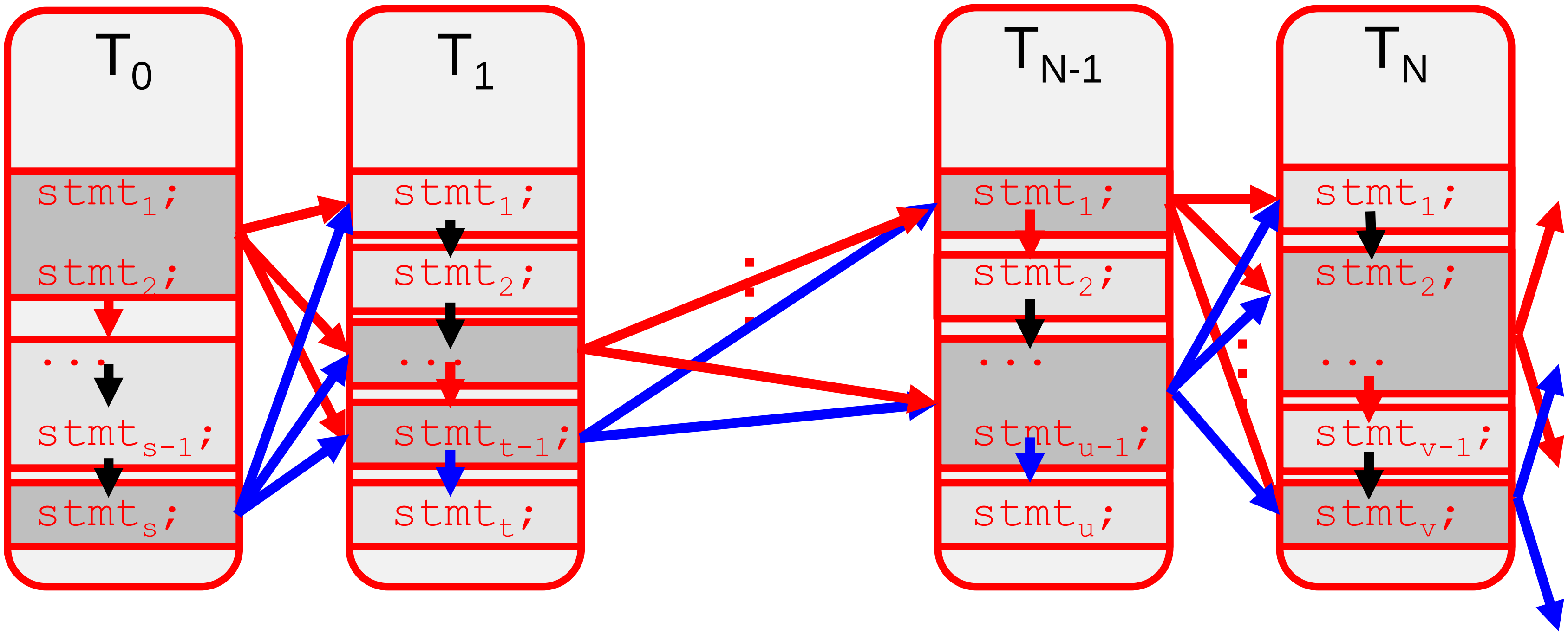
Tiles:



- **tile**: (contiguous) subset of visible statements
  - other tile types possible: random subsets, data-flow driven, …
- **tiling**: partition of program into tiles
- **uniform window tiling**: all tiles have same size
  - number of visible statements

Tiling threads

Tile selection:

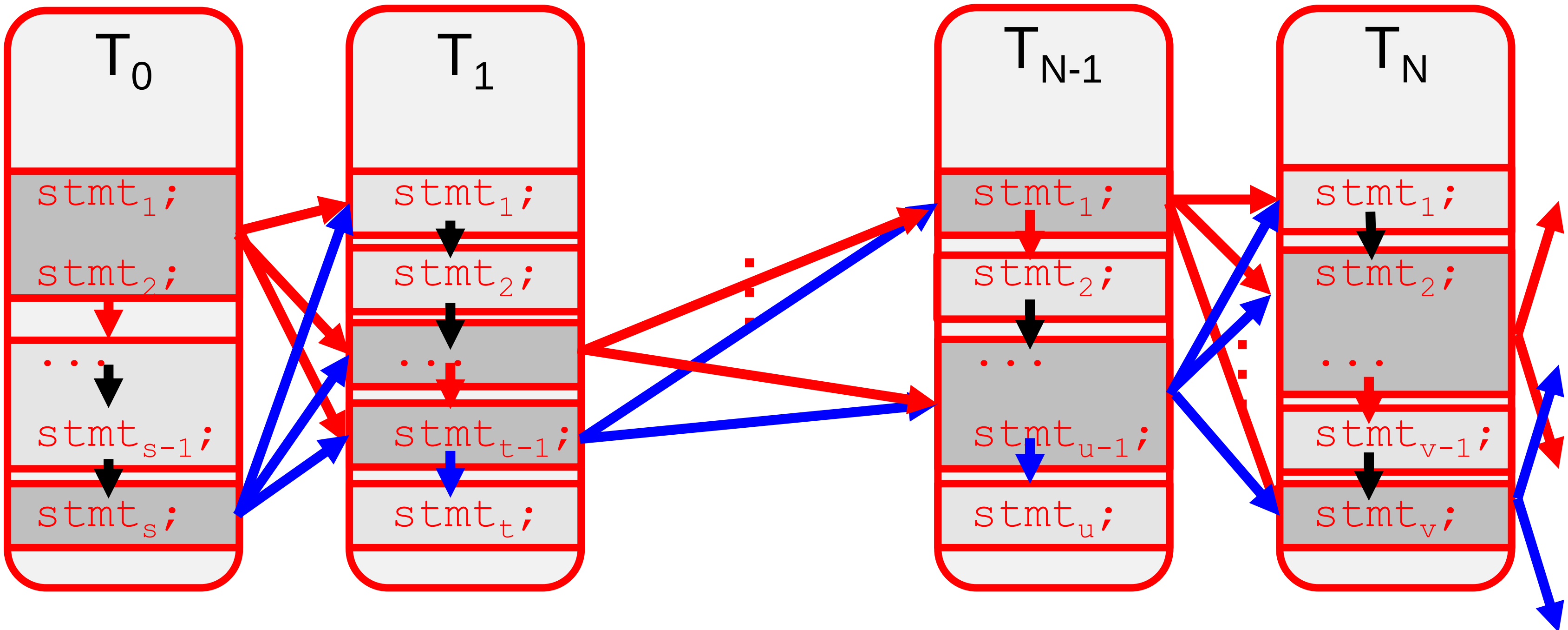


- **z-selection**: subset of z tiles for each thread
  - **context switches** are **only** allowed from **selected tiles**

  $\Rightarrow$ context switches can only go into other selected tiles (or first thread statement)
- each z-selection specifies a **reduced interleaving instance**

Tiling threads

Completeness of selections:

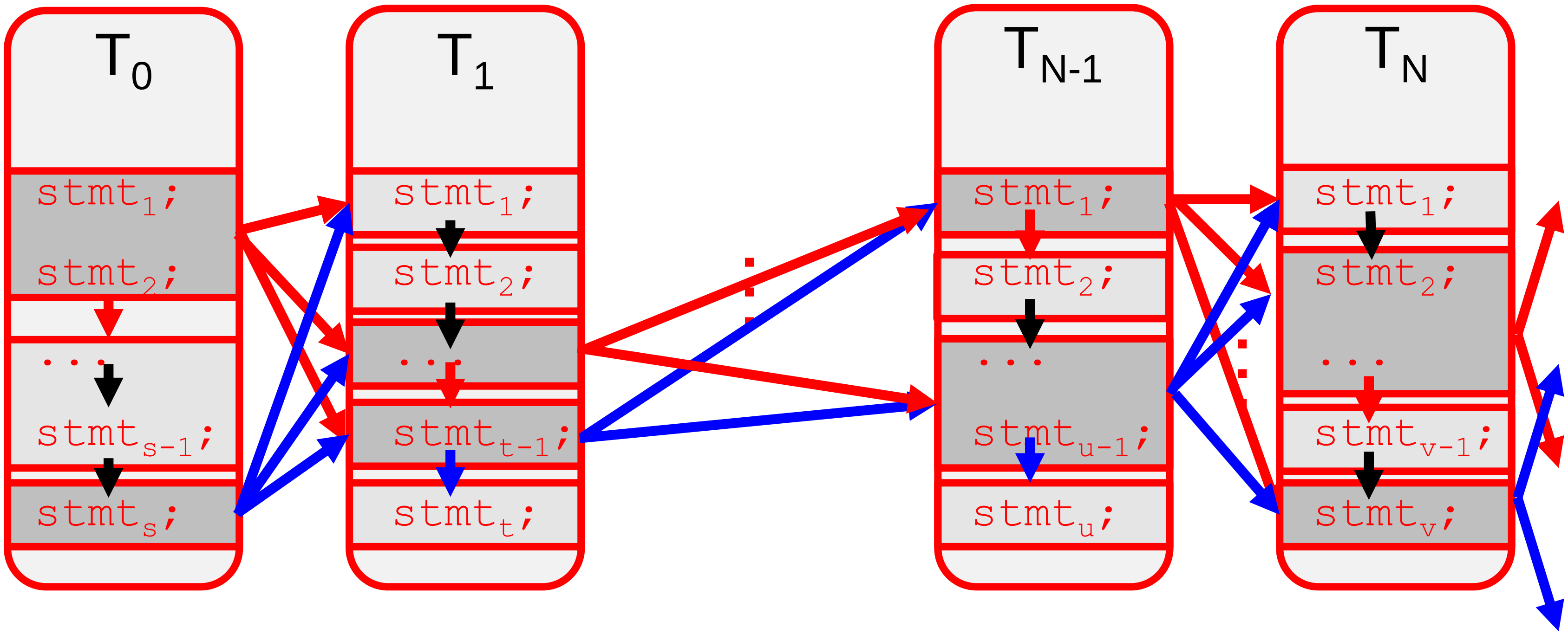


- each interleaving with *k* context switches can be covered by a [*k*/2]-selection $\vartheta \in \Theta_P$
  - each thread can only switch out at most [*k*/2] times

$\Rightarrow$ set of all [*k*/2]-selections together covers all interleavings with *k* context switches: $\mathcal{I}_k(P) = \bigcup_\vartheta \mathcal{I}_k(P_\vartheta)$

Tiling threads

Completeness of selections:



- number of selections grows exponentially

⇒ sampling

VeriSmart

(Verification Smart)

VeriSmart implements swarm verification by task competition for multi-threaded C.

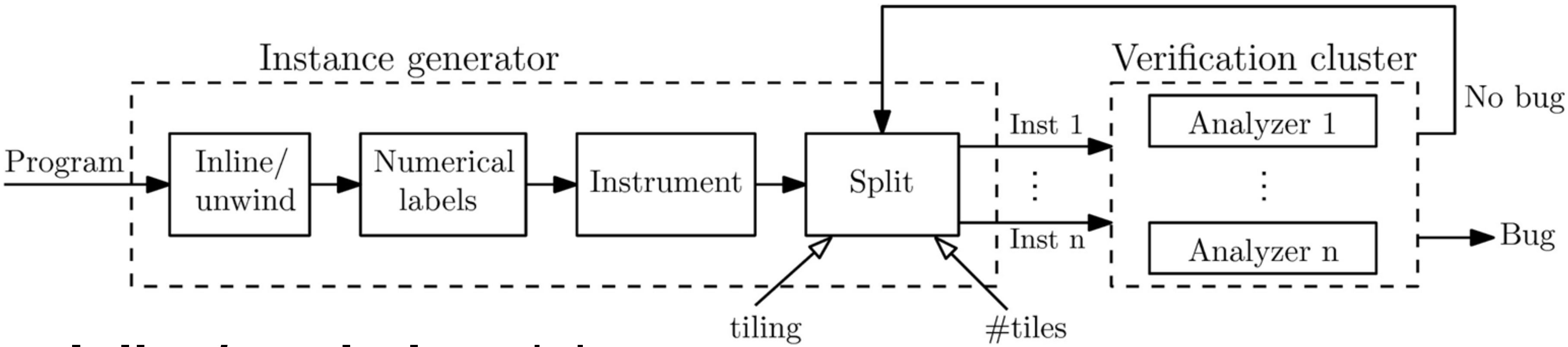
Target:

- C programs with "**rare**" concurrency bugs, i.e.,
  - "**large**" number of interleavings
  - "**few**" interleavings lead to a bug
- **automatic bug-finding** (bounded analysis, not complete)
- **reachability**
  - assertion failure
  - out-of-bound array, division-by-zero, …
  - linearizability → reachability [Bouajjani et al., POPL'15, ICALP'15]

Approach:

- source-to-source translation to generate instances (for tiling)
  - instances are bounded concurrent programs
- use cluster to run Lazy-CSeq over instances [Inverso et al., CAV'14]

VERISMART architecture



- **Inline/unwind** module:
  - concurrent program ⟶ bounded concurrent program
- **Numerical labels** module:
  - inject numerical labels at each visible statement
- **Instrument** module:
  - instrument the code with guarded commands (**`yield`**) that can enable/disable context switch points at numerical labels
- **Split** module:
  - generate variants with configuration from tiling and #tiles
  - randomize number of generated variants when #variants is large

Why does this work?

Remember:

Each P_ϑ allows only a (small) subset of P's interleavings

We assume bugs are rare,

- so for most ϑ, P_ϑ does not exhibit the bug…
- … and the analysis will run out of time
- but if P_ϑ does exhibit the bug…
- … the analysis will find it quick(er)

Hence,

- overall CPU time consumption goes (way) up…
- … but with enough cores CPU time is free and…
- mean wall clock time to find failure goes down

Experimental Evaluation

ABA problem

- Concurrency problem that can occur when a thread reads a shared variable twice and another thread accesses it between the two reads

- Sequence of events that identify an ABA problem:
  - Thread 1 reads A from shared memory, and then is preempted
  - Thread 2 modifies this value from A to B and then back to A
  - Thread 1 is resumed and perceives that nothing has changed

Why is ABA a problem?

- Thread 1 acts as nothing has changed but …

  …… this is not the case and can lead to error

- For example, in lock-free data structures:
  - an item X is removed from the structure
  - a new item Y is added to the structure
  - due to optimizations, it is common that Y is allocated at the same location as X
  - pointer to Y is same as pointer to X
  - however Y might be different from X

Experiments on lock-free data structures

`eliminationstack`:

- ABA problem: requires 7 threads for exposure
- Lazy-CSeq can find bug in **~13h** and **4GB**
  - #unwind=1, #rounds=2, #threads=8, #visible=52
- all other tools fail

`safestack`:

- ABA problem: requires context bound of 5
- Lazy-CSeq can find bug in **~7h** and **6.5GB**
  - #unwind=3, #rounds=4, #threads=4, #visible=152
- all other tools fail

eliminationstack: Results

- Lazy-CSeq: 46764 sec, 4.2 GB
- CBMC (sequential): 80.8 sec, 0.7 GB
  - average over 3000 interleavings, bug not found

fastest instances very fast – 1000x

reduced memory consumption – 4x

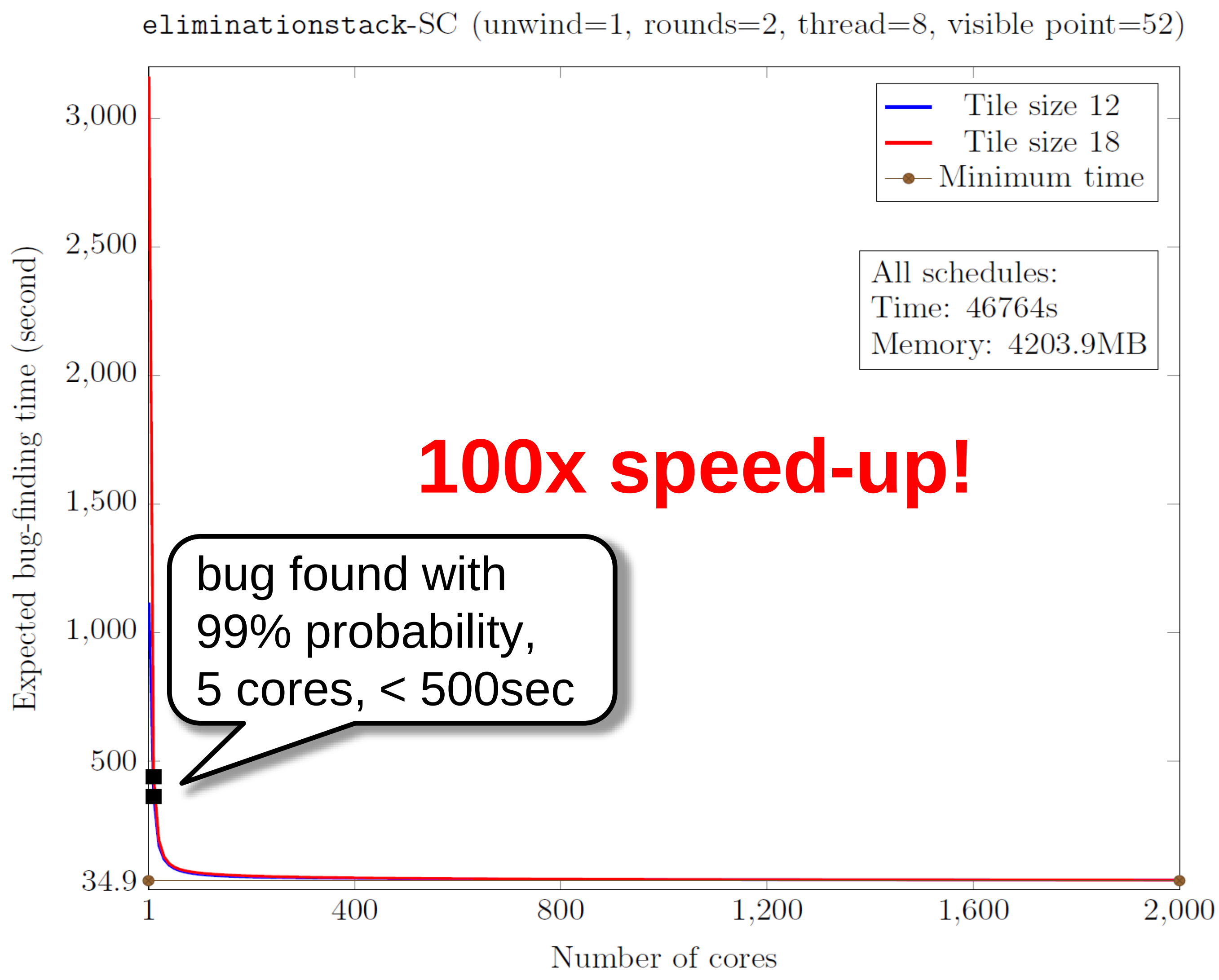
ERISMART: 2 til								
#1: tile si			#2: tile size 14, t			size 18, t_max 3hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	34.9	945.2	Min	39.7	979.84	Min	37.1	999.8
Max	4753.6	1199.1	Max	7195.2	1281.3	Max	10762.0	1785.5
Average	1116.3	1017.8	Average	2169.5	1096.3	Average	3162.41	1156.91
instances with bug: 38.33%			instances with bug: 61.38%			instances with bug: 69.01%		

average still very fast – 40x

high fraction of bug-exposing instances

some slowdown for larger tile sizes – 10x

eliminationstack:
Expected bug finding time



safestack (SC): Results

- Lazy-CSeq: 24139 sec, 6.6 GB
- CBMC (sequential): 55.4 sec, 0.7 GB
  - average over 3000 interleavings, bug not found

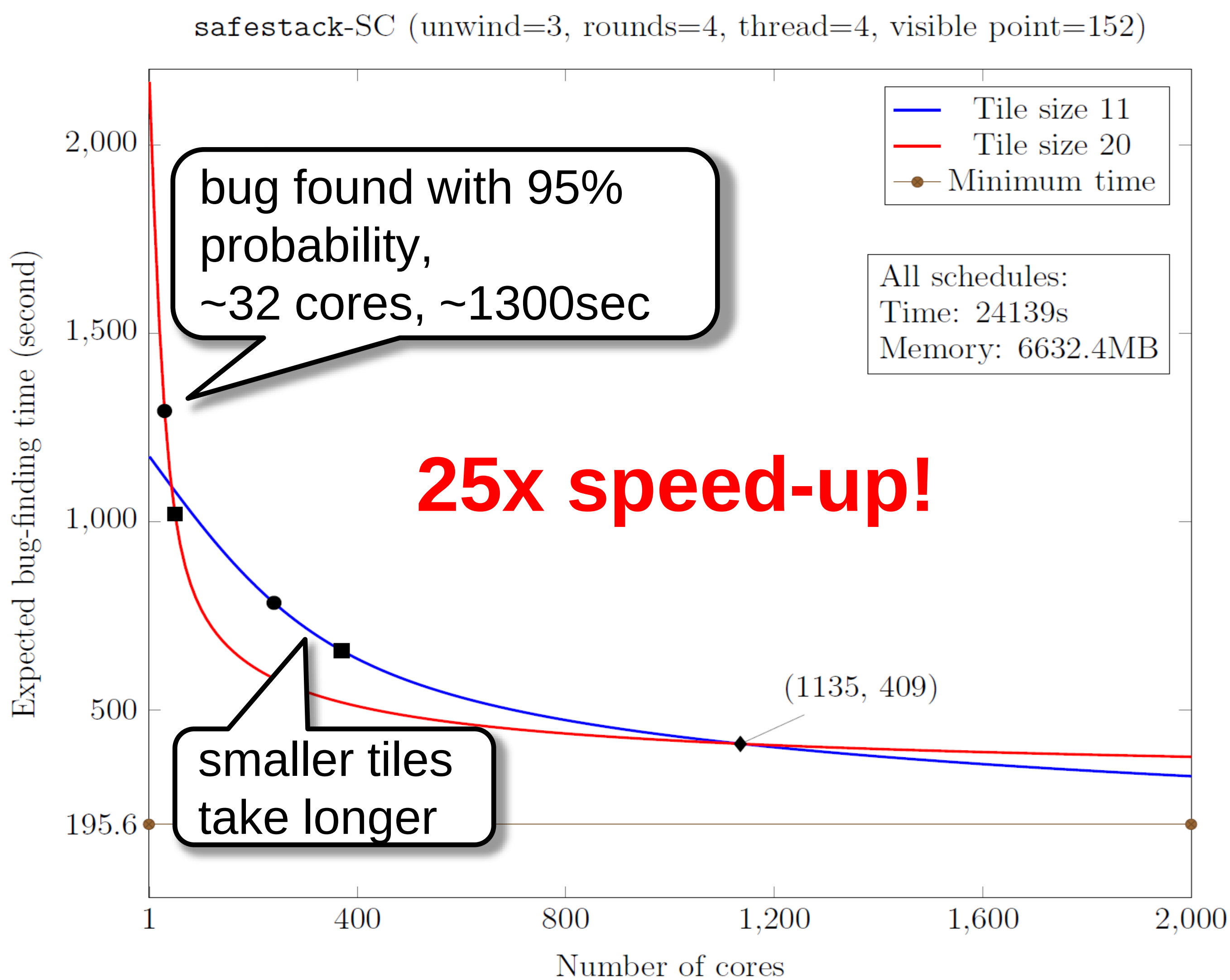
VERISMART: 4 tiles per thread								
#1: tile size 11, t_max 1hr			#2: tile size 14, t_max 1hr			#3: tile size 20, t_max 4hrs		
Verification	Time	Memory	Verification	Time	Memory	Verification	Time	Memory
Min	195.6	774.5	Min	574.8	846.6	Min	313.0	850.3
Max	2662.6	1265.7	Max	3521.8	1450.4	Max	10315.8	3830.8
Average	1172.2	928.8	Average	1851.1	1147.3	Average	2167.5	1230.1
instances with bug: 1.26%			instances with bug: 2.14%			instances with bug: 10.20%		

lower fraction of bug-exposing instances than eliminationstack

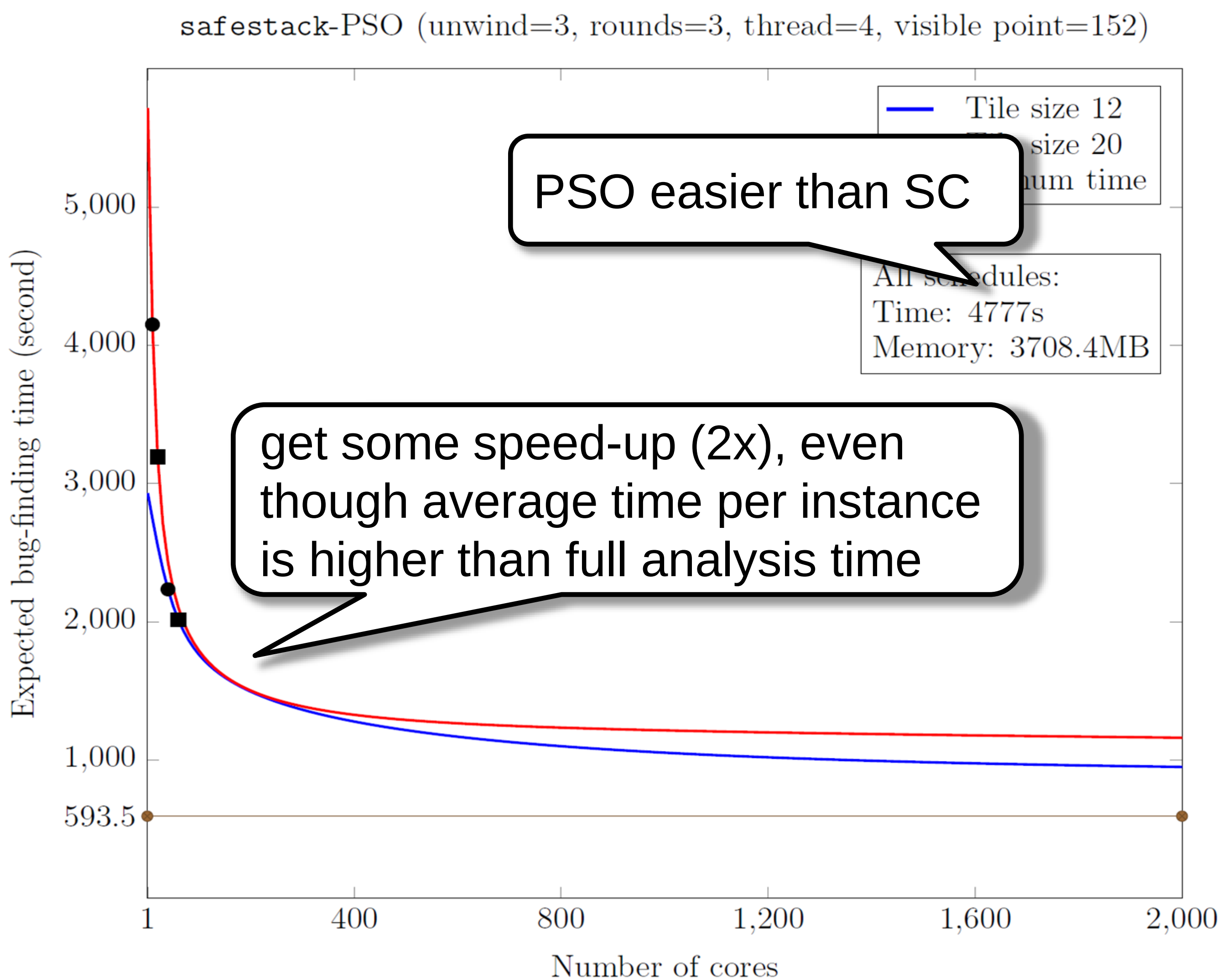
…but boosted with larger tile sizes

$\Rightarrow$ similar picture, but less advantage for VERISMART

safestack (SC):
Expected bug finding time



safestack (PSO):
Expected bug finding time



Conclusions

- first task-competitive swarm verification approach
- exploits availability of many cores to reduce mean wall clock time to find failure
  - allows us to handle very hard problems
  - high speed-ups already for 5-50 cores
- reduced interleaving instances boost bug-finding capabilities

Future Work

- production-quality implementation based on LLVM
- other backends (testing)
- other tiling styles
- fast over-approximations to filter out safe instances

References

- Parallel Bug-finding in Concurrent Programs via Reduced Interleaving Instances

  [Nguyen-Schrammel-Fischer-La Torre-Parlato, ASE'17]

- Swarm verification techniques

  [Holzmann-Joshi-Groce, IEEE Trans. Software Eng., 2011]

- Swarm verification [Holzmann-Joshi-Groce, ASE'08]

- Cloud-based verification of concurrent software

  [Holzmann, VMCAI'16]