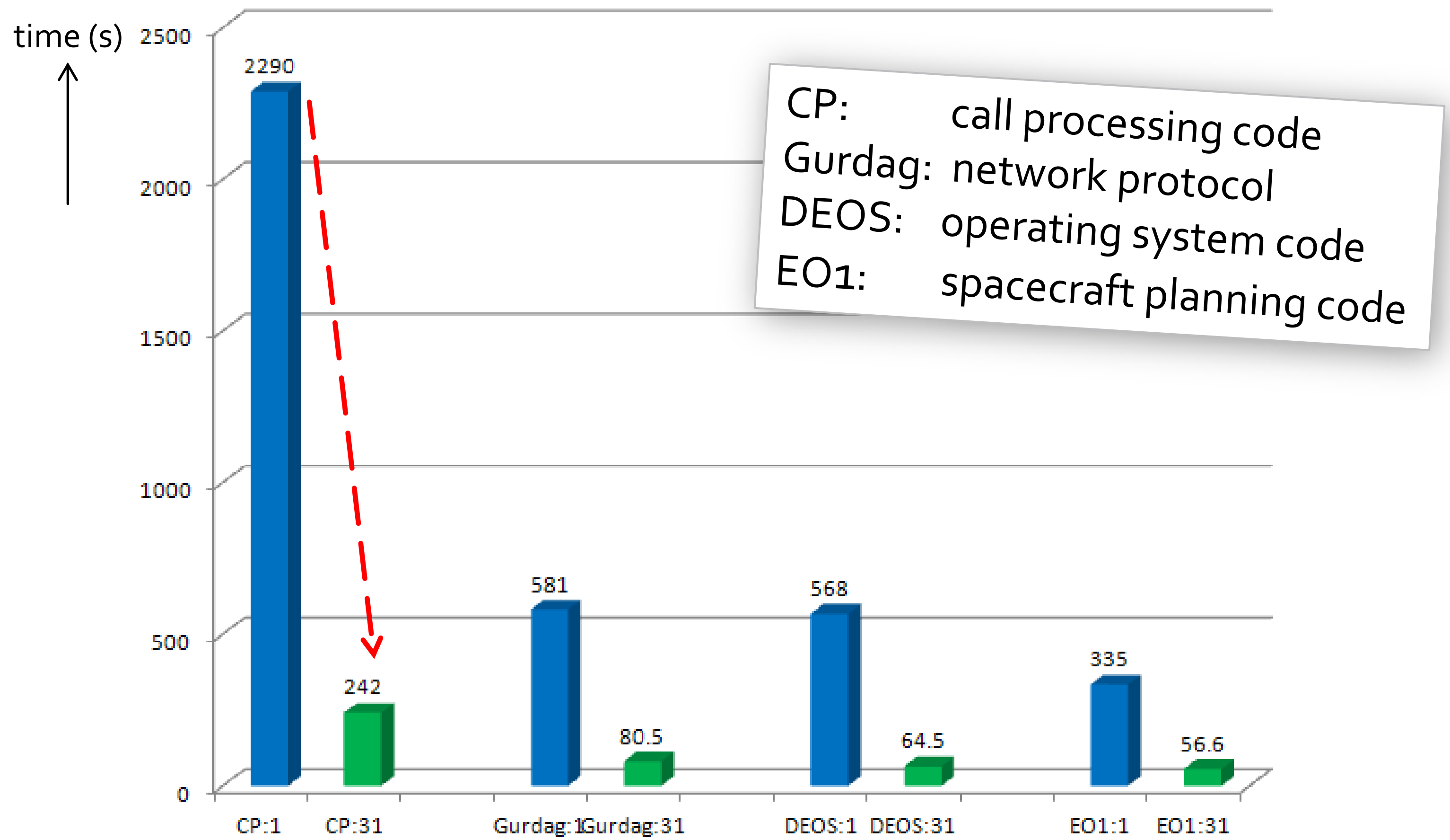
Speeding up the Analysis of Complex Software with Parallel Model Checking

gerard holzmann
lab for reliable software
nasa/jpl

LARS
NASA/JPL Laboratory for Reliable Software

what would it take to achieve speedups like these?



parallel breadth-first search

PROS

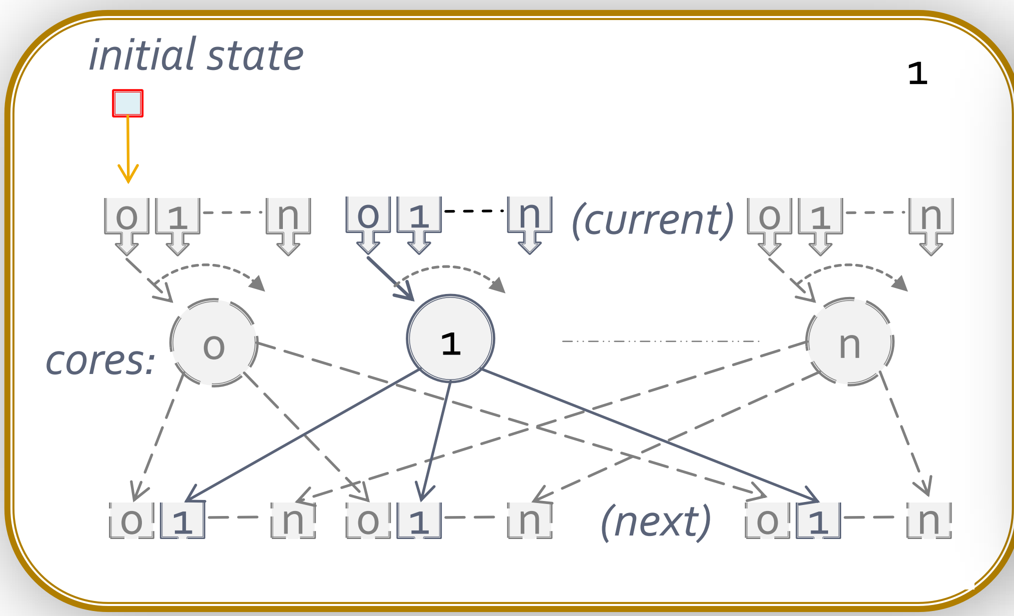
- simplest search mode for a logic model checker
  - basic reachability analysis
- there is no ordering requirement on state exploration
  - relatively easy to parallelize
- always finds the *shortest* counter-example first

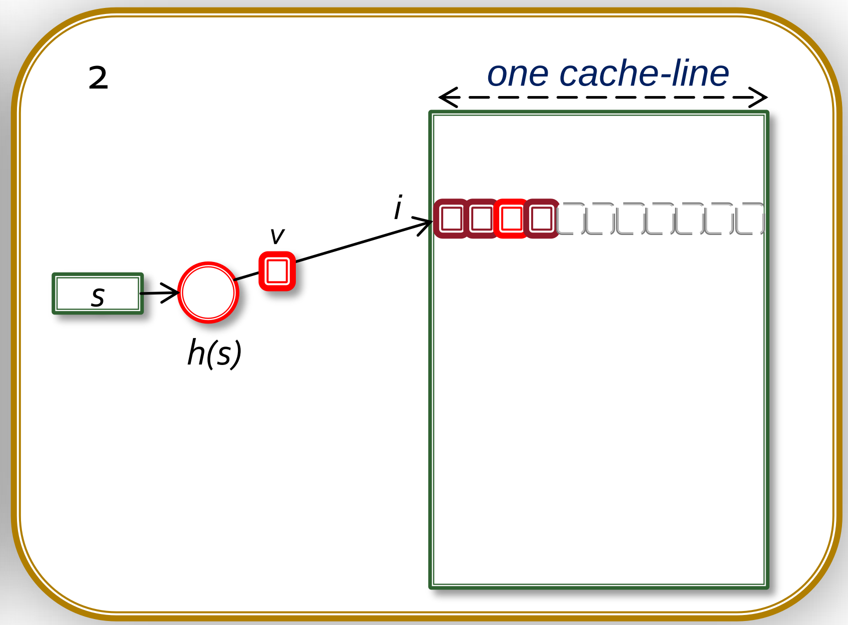
CONS

1. parallelization requires locks and synchronization
   - which can limit performance
2. often requires more memory than a depth-first search
3. traditionally restricted to the subclass of LTL defining *safety* properties
   - invariants, absence of assertion violations, absence of deadlock, etc.

1: eliminate locks

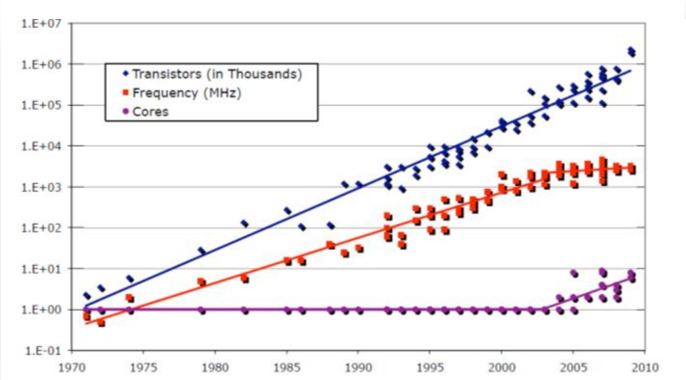
- design a lock-free algorithm
  1. lock-free & contention-free queues
  2. a modified cache-aware hash-table





2: requires (modestly) more memory

- both the *number of cores* and *the size of RAM* grows with Moore's curve: i.e., exponentially fast
  - but clock-speeds remain constant



source: Olukotun, Hammond, Sutter, Smith, Batten & Asanovic

- this means:
  - memory is *not* the bottleneck
  - performance is linked to clockspeeds ***unless*** we exploit parallelism

3: traditionally restricted to safety

safety: p is invariant
liveness: [] (p -> <> q) – *when p occurs, eventually q will occur as well*

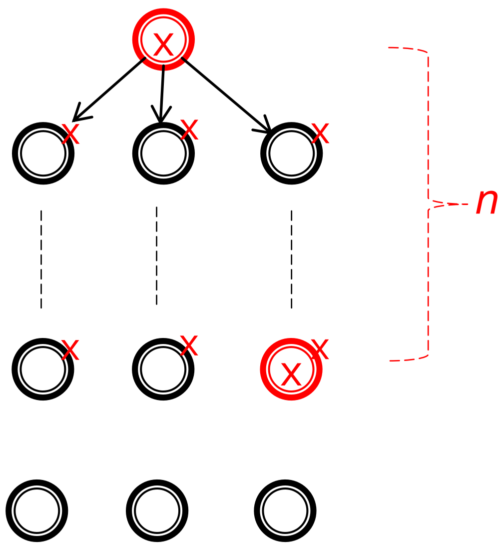
safety properties can be checked with a breadth-first search
liveness properties are harder:
they require a *cycle detection* algorithm
this can be done efficiently with a *depth*-first search
at up to *twice* the cost of a standard depth-first search

the best known methods for verifying **liveness** with a breadth-first search carry excessive overhead:
the cost becomes *quadractic*, for instance (*R*=size of graph)
if $R = 5.10^6$ then $2.R = 10.10^6$, but $R^2 = 25.10^{36}$
if *R* takes *2 seconds*, *2.R* takes 4 sec, and R^2 takes *10^7 seconds* (*11 days*)

3: *bounded search*

a "piggyback" algorithm

$[\,]\,(p \rightarrow <>_n\, q)$	bounded liveness
when p happens, q will happen as well *within n steps*	
$<>(p \wedge [\,]_n\, !q)$	signature of counter-examples



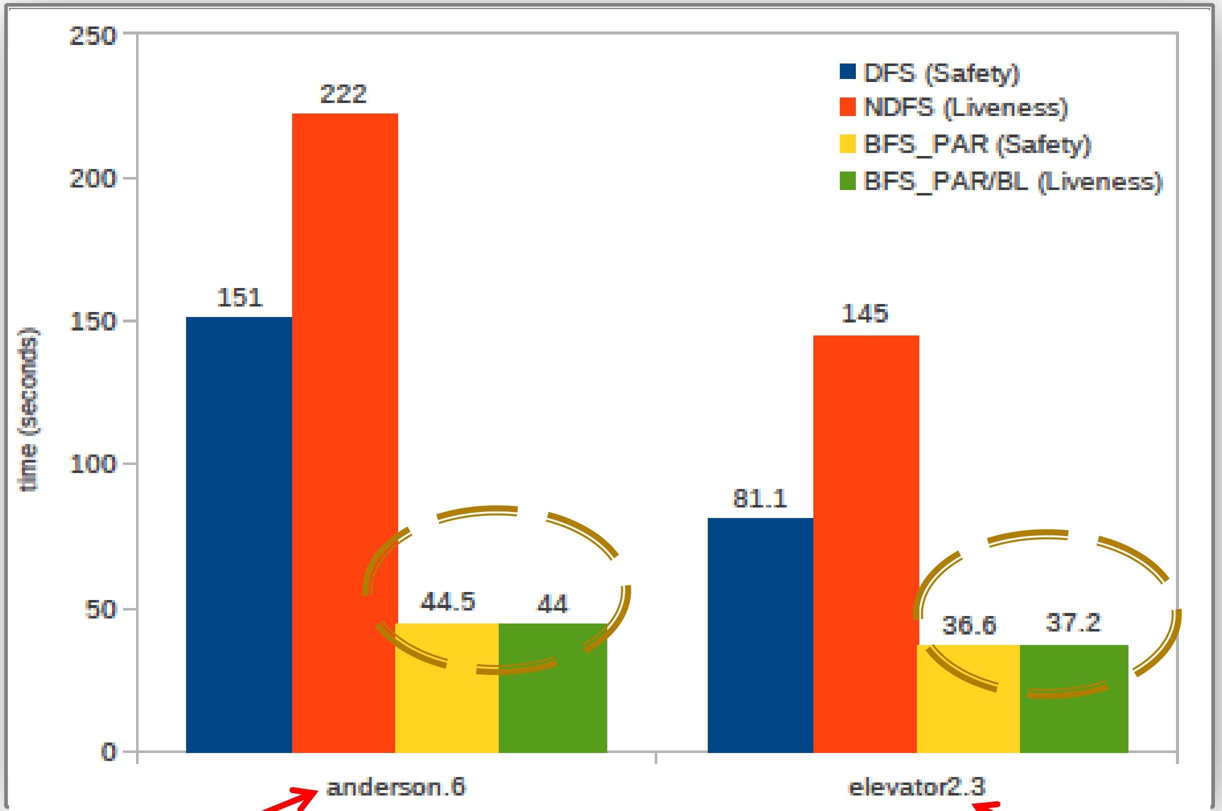
"piggyback search"

bounded search
we perform a check on paths of max length *n*

PRO:
- simple to implement
- adds a small *constant* memory overhead for propagating *tags*, but adds *virtually no time*
- the cost is: *c.R* with 1< *c* <<2

CON:
to limit memory overhead, we carry only 1 *tag field*
this means we can miss counter-examples: we accept a small chance of incompleteness
remarkably: the algorithm works almost always

bfs liveness – two examples
piggyback algorithm



49 million states
reports liveness violations
[](<>(P[2]@CS))

27 million states
no liveness violations (worst case search)
[] ((!((req[0]==1))) || ((!((p==0))) U (((p==0)) U ((!((p==0))) U (((p==0)) U (((p==0)) && ((cabin@open)))))))))

synopsis

- the new parallel breadth-first search option, with the piggyback algorithm, delivers a remarkable increase in performance on multi-core systems
  - allowing us to tackle more complex software verification problems (medical, automotive, aerospace)
  - the extension described here will become part of Spin version 6.2.0
  - a paper "*Parallelizing the Spin Model Checker*" was submitted for publication