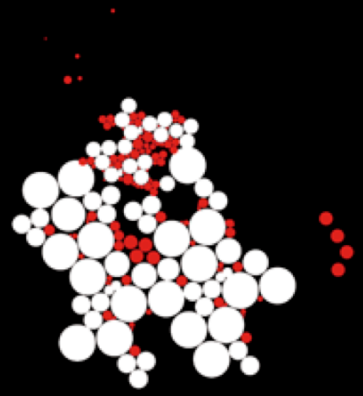
UNIVERSITY OF TWENTE.



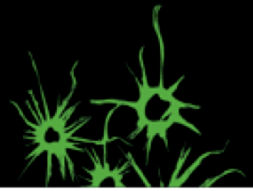
SHARED HASH TABLES IN PARALLEL MODEL CHECKING

IPA LENTEDAGEN 2010

ALFONS LAARMAN

JOINT WORK WITH MICHAEL WEBER

AND JACO VAN DE POL



23/4/2010

AGENDA

- Introduction
  - Goal and motivation
  - What is model checking?
  - Hash tables
- Related work
- Lockless hash table
- Experiments

GOAL AND MOTIVATION

Goal:

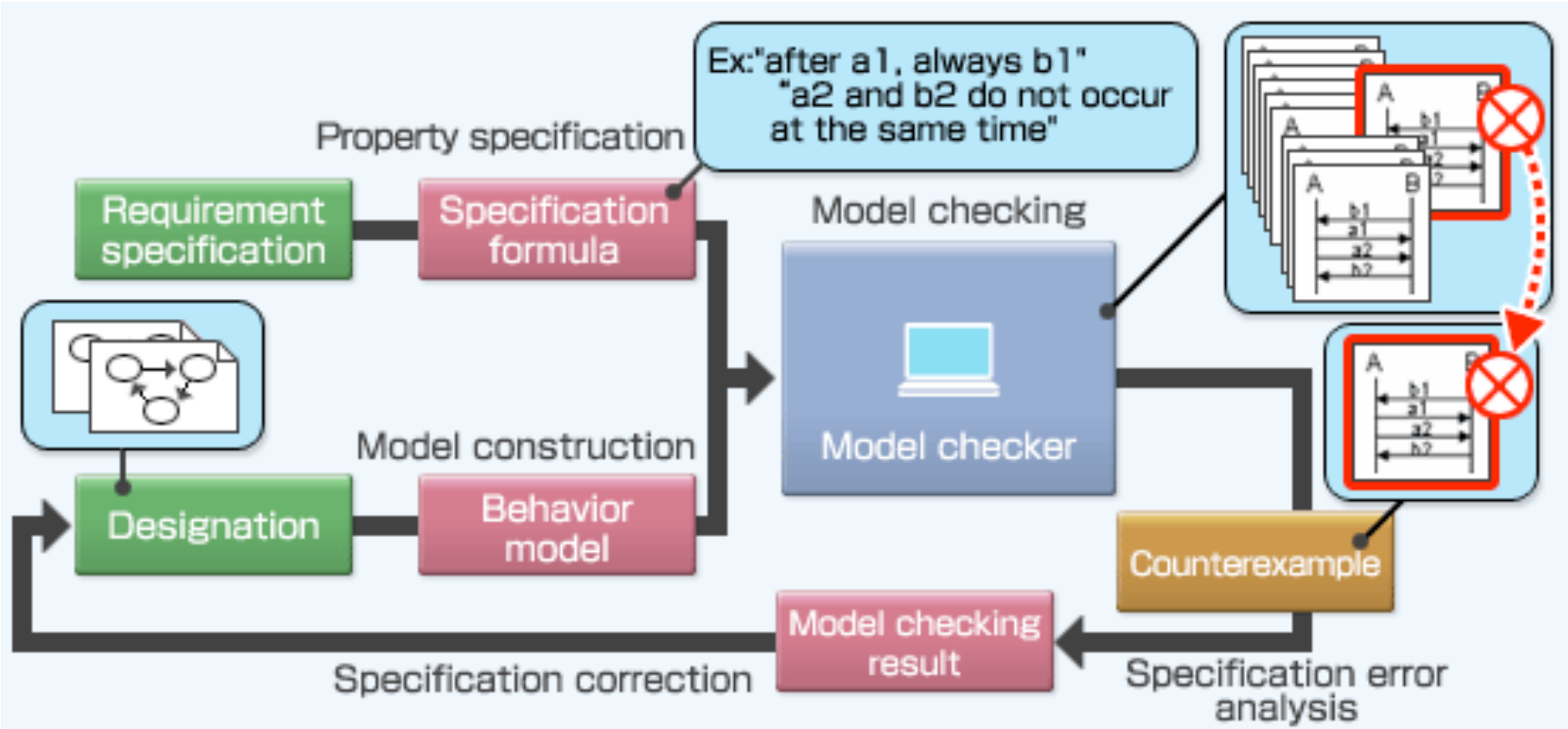
- Realize efficient multi-core reachability

Motivation:

- Multi-core is a necessity
- Reachability is a the basis of many verification problems
- Current model checkers do not scale as good as possible
- If you cannot parallelize reachability efficiently, then how do you parallelize more complicated algorithms:

  full LTL model checking, symbolic reachability, POR, etc

WHAT IS MODEL CHECKING?



www.toshiba.co.jp

WHAT IS MODEL CHECKING?

$Process_1$	$Process_2$
1 x ← x + 1;	1 y ← y + 1;
2 **if** y ==*1* **then**	2 **if** x ==*1* **then**
3 x ← 10;	3 y ← 10;
4 **end**	4 **end**

- PROMELA (SPIN)
- DVE (DiVinE)
- .NET (MoonWalker)
- C/C++ (terminator)
- Process algebraic (mCRL 1/2)
- Timed (UPPAAL)
- Hardware model checkers

WHAT IS MODEL CHECKING?

```
Data: Buffer T = {s_0}, Set V = ∅
1  while state ← T.get() do
2  |  count ← 0;
3  |  for succ in next-state(state) do
4  |  |  count ← count + 1;
5  |  |  if V.find-or-put(succ) then
6  |  |  |  T.put(succ);
7  |  |  end
8  |  end
9  |  if 0 == count then
10 |  |  //DEADLOCK, print trace..
11 |  end
12 end
```

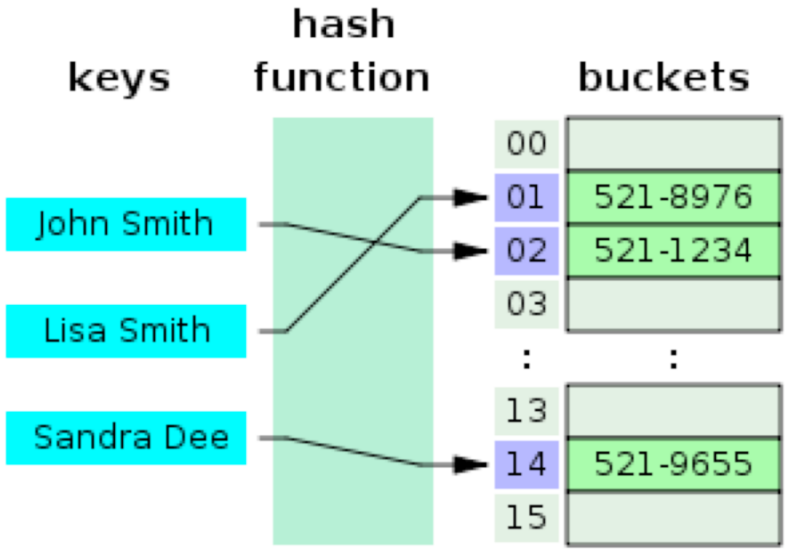
- States are arrays with variables
- Search in state space graph
- V stores all seen states
- DFS or BFS depending on T
- Deadlocks and invariants

Parallelization:

- High throughput
- Synchronization points

HASH TABLES

A key is associated with data by using its hash as an index in a table



(wikipedia)

Hash collisions:

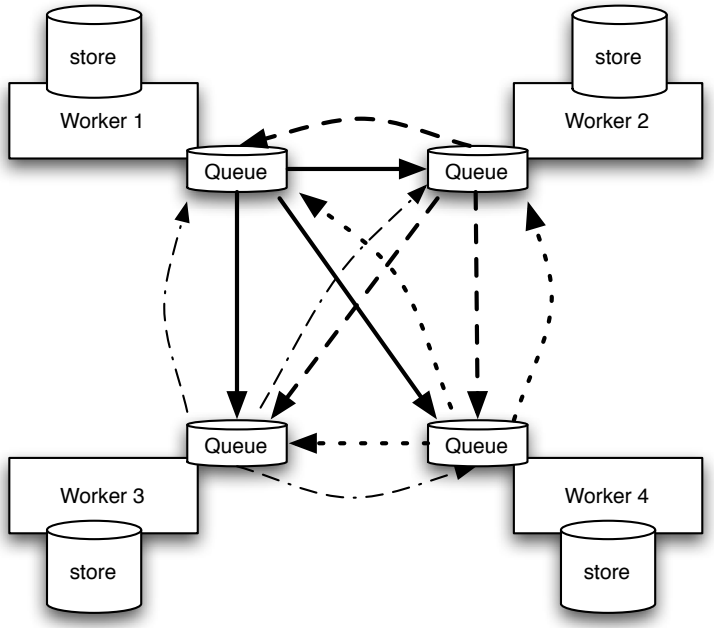
- Create overflow list (chaining) ← large memory working set
- Continue probing (open addressing) ← asymptotic behavior when full
  - Linear probing, double hashing

RELATED WORK

Fast model checkers:

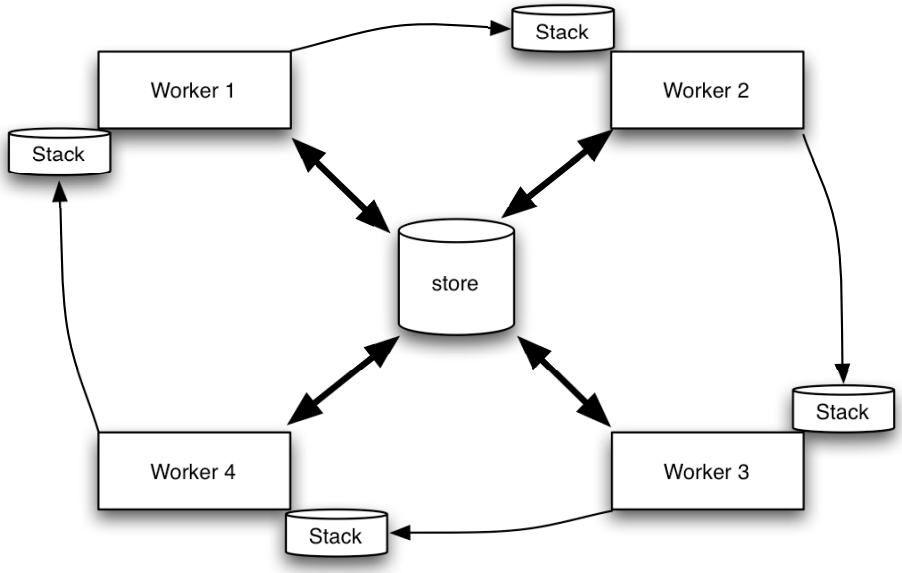
- SPIN
- DiVinE 2.2
- DiVinE with Shared storage

RELATED WORK



DiVinE 2.2: static partitioning

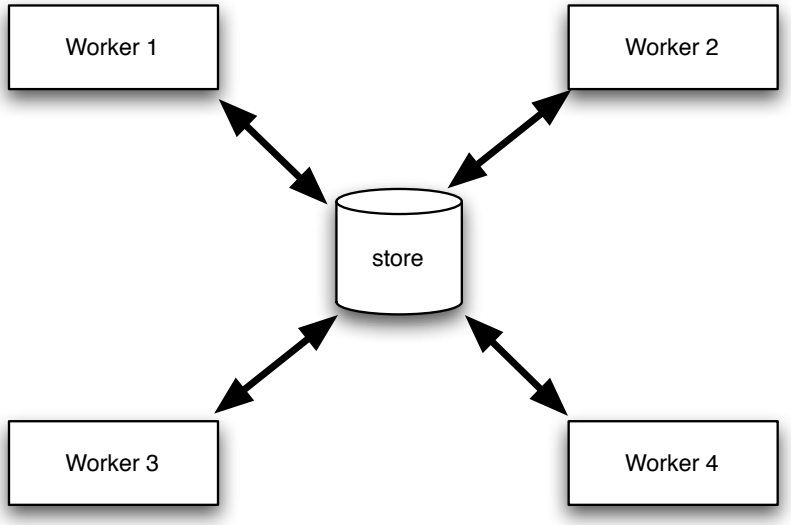
BFS only, high comm. Costs, static load balancing



SPIN 5.2.4: shared storage + stack slicing

DFS only, multiple sync. points, specific case of load balancing

RELATED WORK



Barnat, Ročkai (2007) Shared hash tables in parallel model checking

- “Shared hash tables do not scale beyond 8 cores”
- “Could not investigate lockless hash table solution”

- Flexible reachability algorithm
- Flexible load-balancing

LOCKLESS HASH TABLE

- Design
- Lockless hash table
- Load balancing

LOCKLESS HASH TABLE

•Investigate requirements on shared storage

•Investigate hardware (cache behavior)

Requirements

•Find-or-put operation only

•Scale by keeping a low memory working set

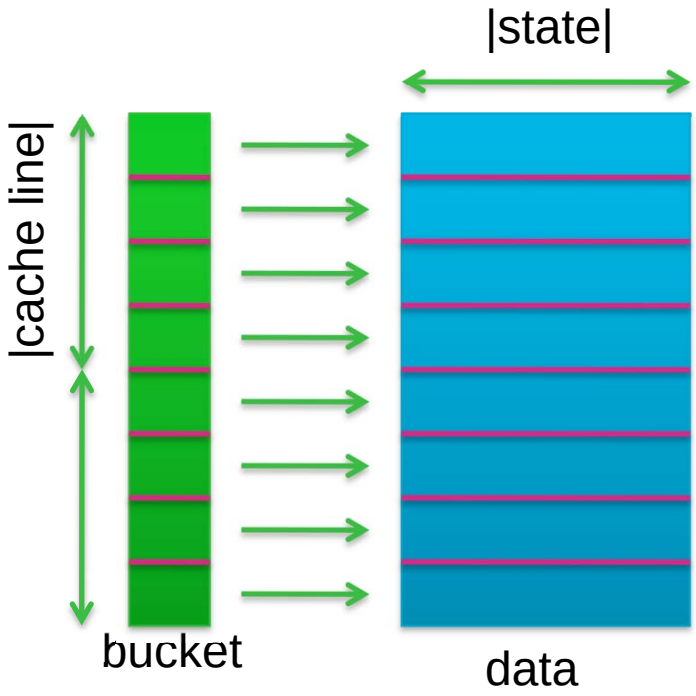
•No pointers, no allocation

•No resize!

•Statically sized state vectors

LOCKLESS HASH TABLE

- Open addressing
- Hash memoization
- Separate data
- Walking the line
- Lockless (CAS + write bit)



See also Cliff Click JavaOne talk (2007)

LOCKLESS HASH TABLE

```
Data: size, Bucket[size], Data[size]
input  : vector
output: seen
1  num ← 1;
2  hash_memo ← hash_num(vector);
3  index ← hash  mod size;
4  while true do
5  |  for i in walkTheLineFrom(index) do
6  |  |  if empty = Bucket[i] then
7  |  |  |  if CAS(Bucket[i], empty, ⟨hash_memo, write⟩) then
8  |  |  |  |  Data[i] ← vector;
9  |  |  |  |  Bucket[i] ← ⟨hash, done⟩;
10 |  |  |  |  return false;
11 |  |  |  end
12 |  |  end
13 |  |  if hash_memo = Bucket[i] then
14 |  |  |  while ⟨−, write⟩ = Bucket[i] do ..wait.. done
15 |  |  |  if ⟨−, done⟩ = Bucket[i] ∧ Data[i] = vector then
16 |  |  |  |  return true;
17 |  |  |  end
18 |  |  end
19 |  end
20 |  num ← num + 1;
21 |  index ← hash_num(vector)  mod size;
22 end
```

Walk-the-line
Linear probing

Wait for write in data array to complete

Double hashing

LOAD BALANCING

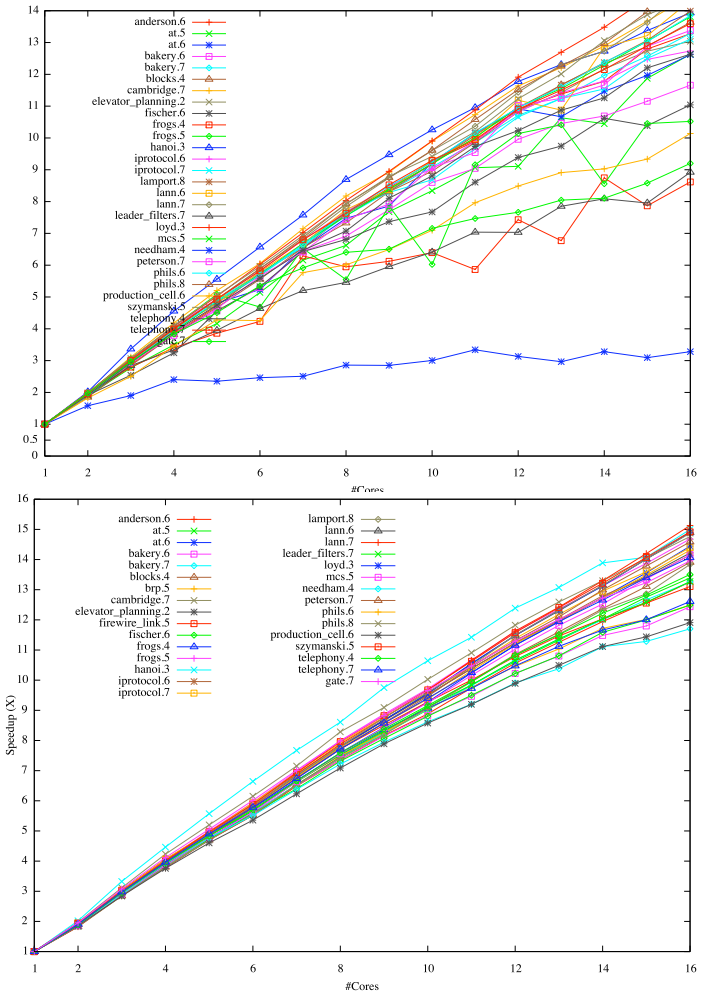
Static load-balancing

Workers can run out of work

Work stealing/handoff

Synchronized random polling

[Sanders97]



SUMMARY

As a summary, we implemented:

- The lockless hash table (in C)
- Reachability DFS + BFS
- Static load-balancing
- Synchronized random polling

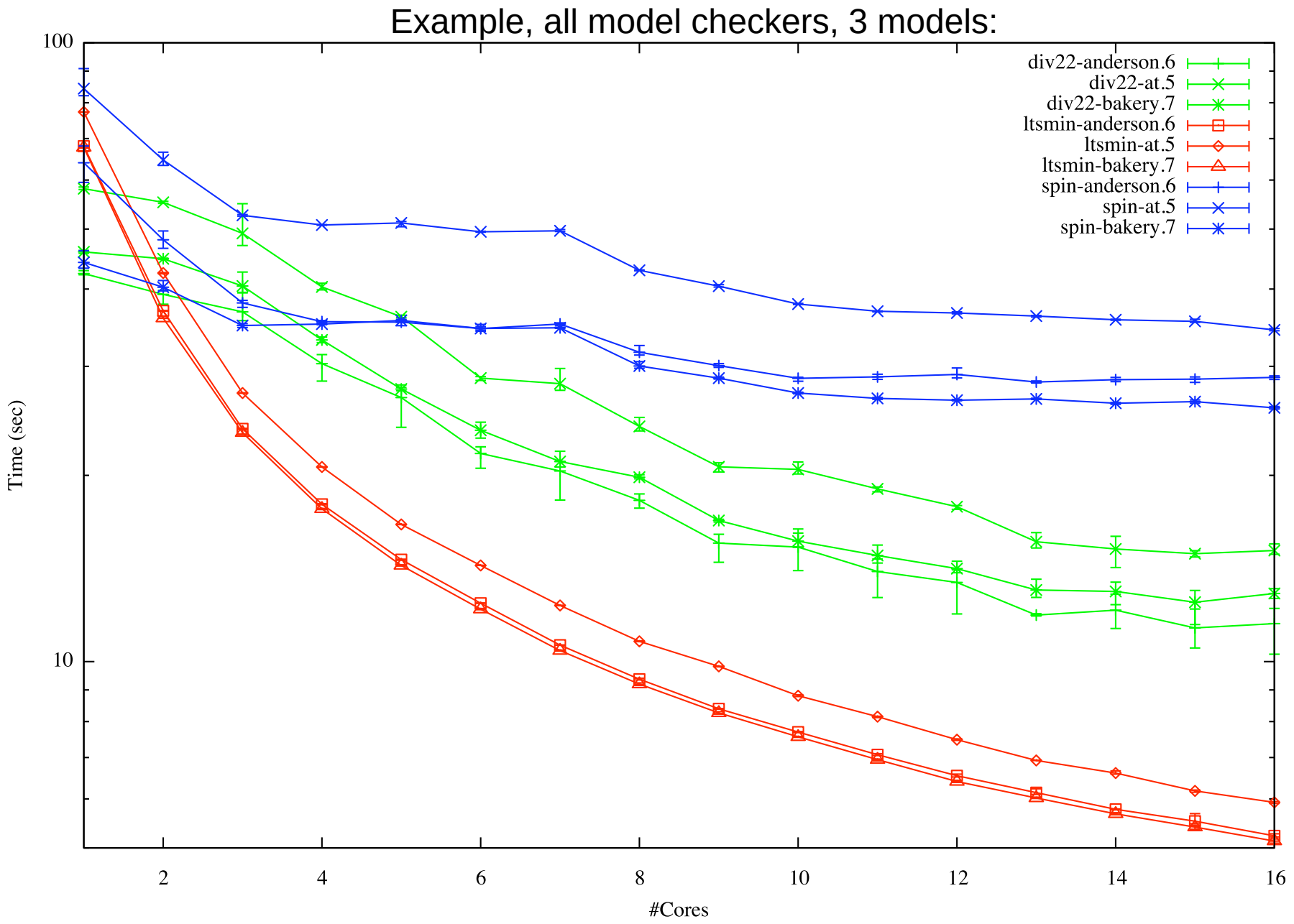
Reused DiVinE next-state function

EXPERIMENTS

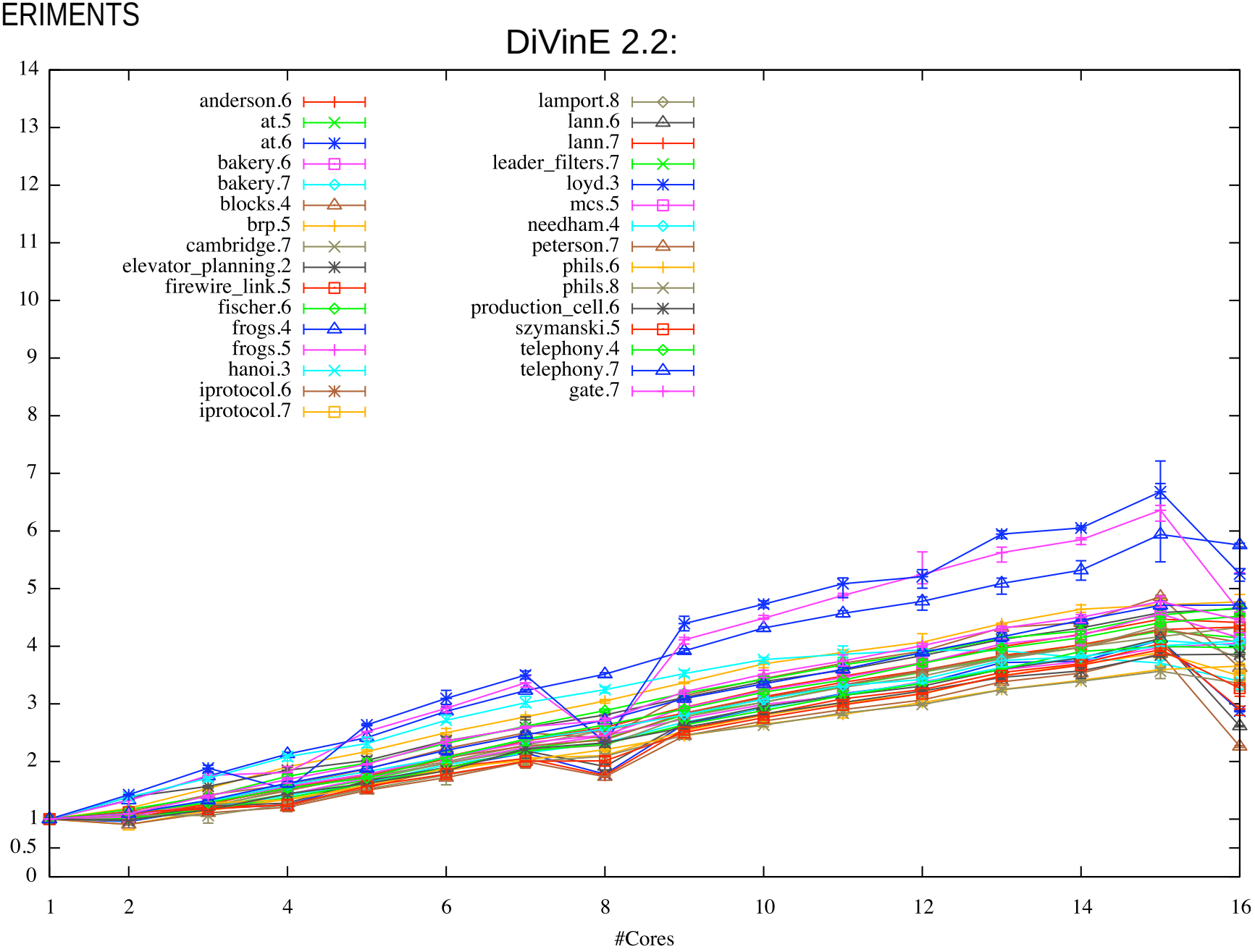
SETUP

- Using CMS 16-way AMD Opteron cluster
- linux 2.6.18, 2.6.32+patch
- 30+ models from BEEM database
  - Translated models for SPIN (same state count!)
- Statically sized hash tables
- Fair

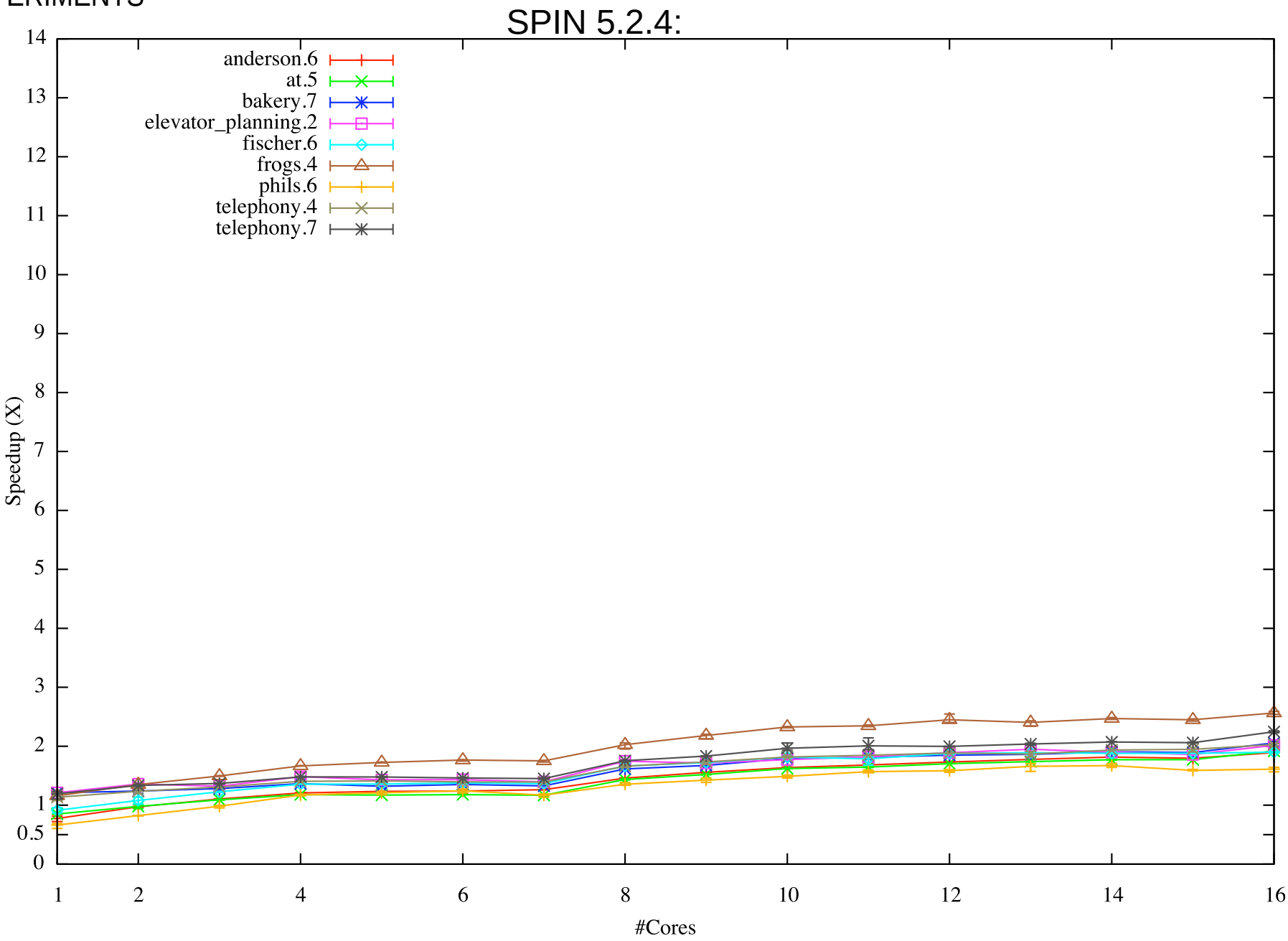
Results →


Example, all model checkers, 3 models:
100
10
Time (sec)
2
4
6
8
10
12
14
16
#Cores
div22-anderson.6
div22-at.5
div22-bakery.7
ltsmin-anderson.6
ltsmin-at.5
ltsmin-bakery.7
spin-anderson.6
spin-at.5
spin-bakery.7

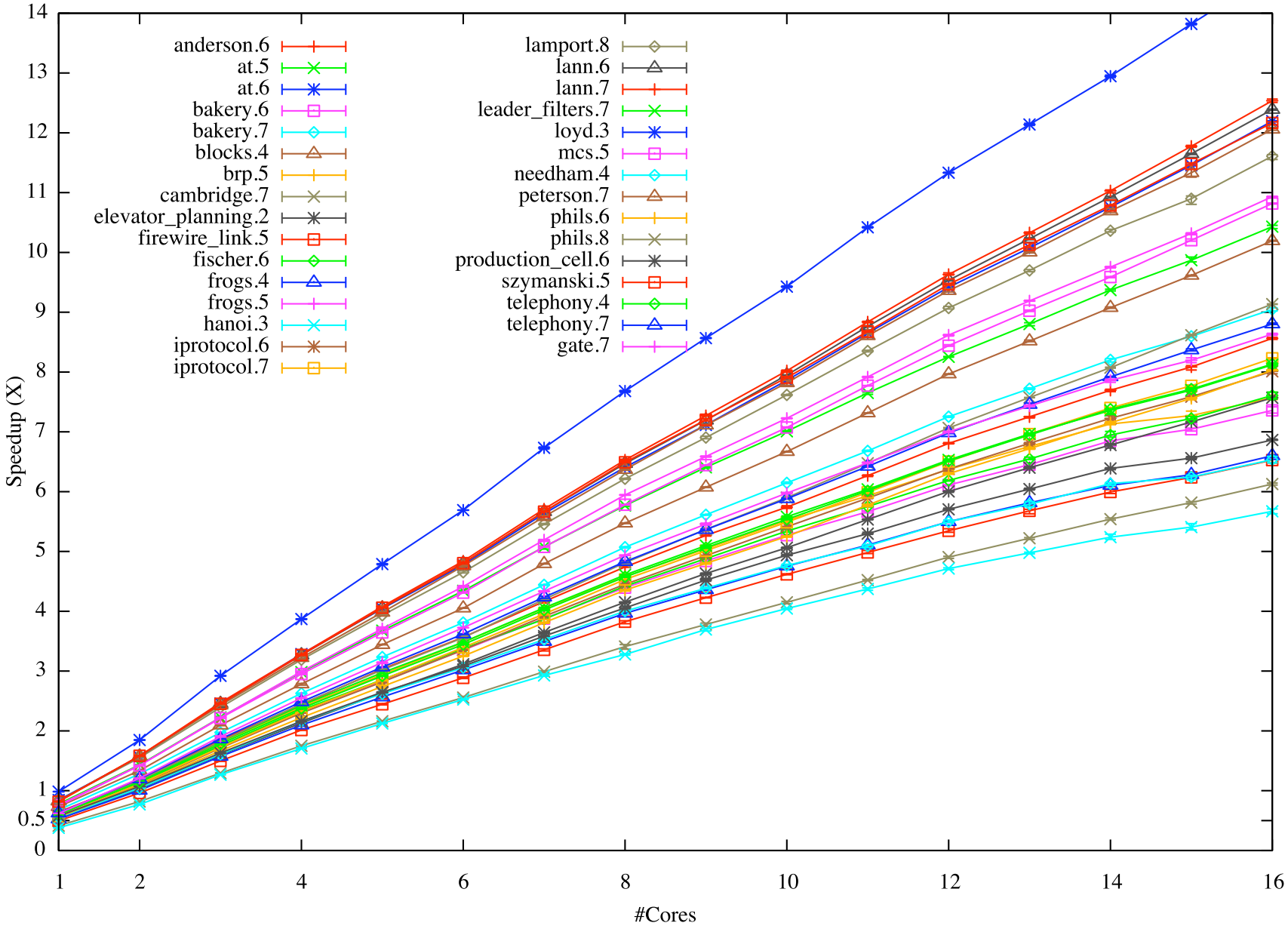
EXPERIMENTS



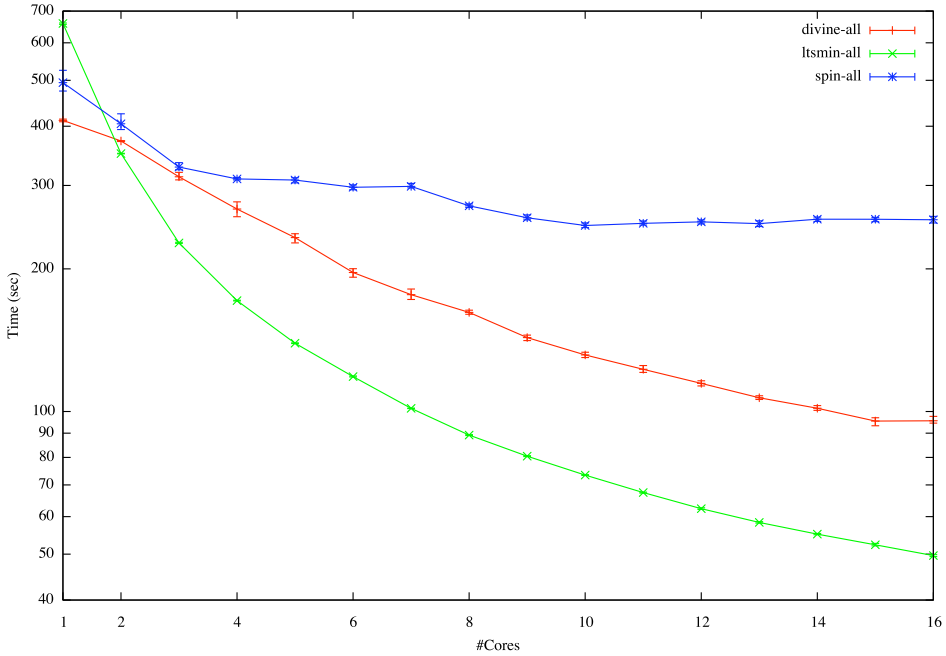
EXPERIMENTS

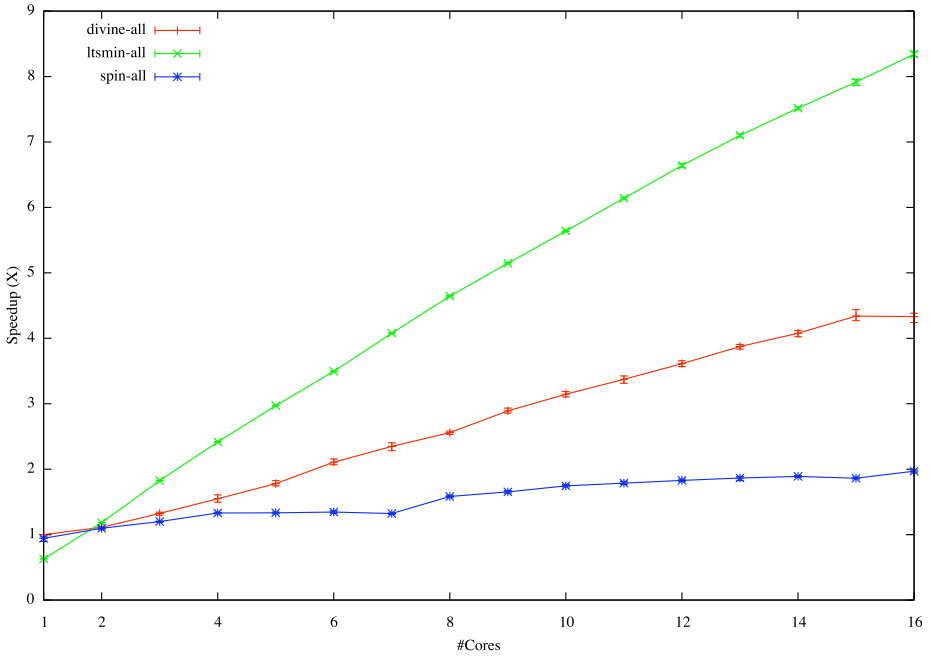


LTSmin (lockless hash table):

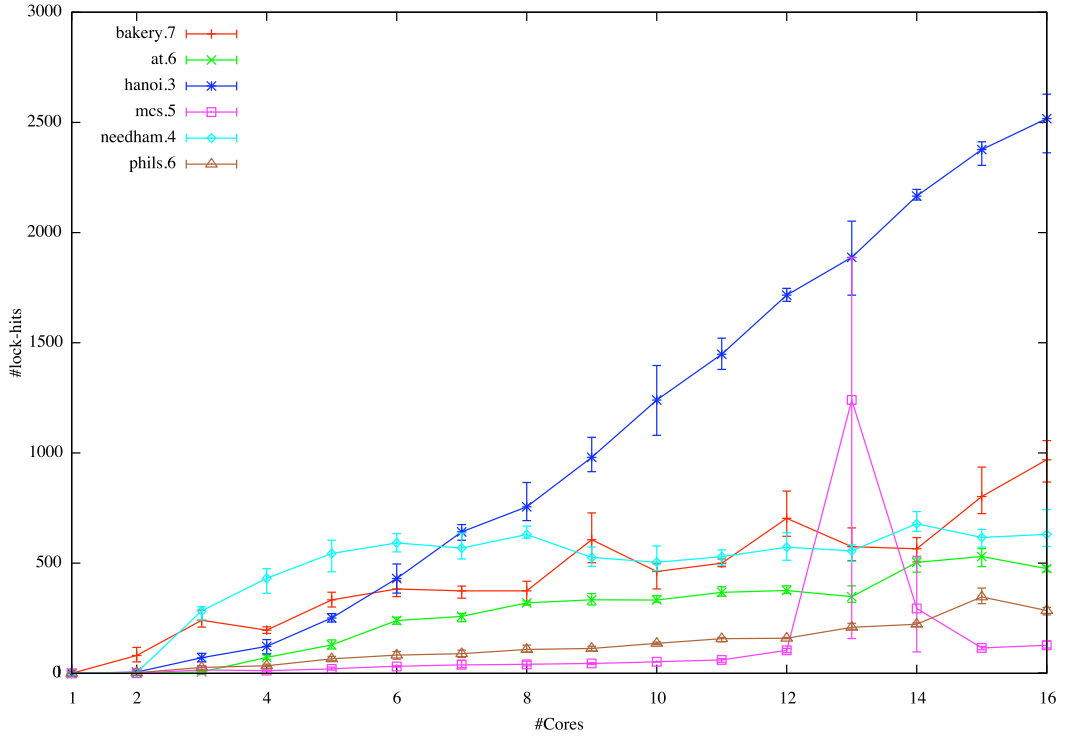


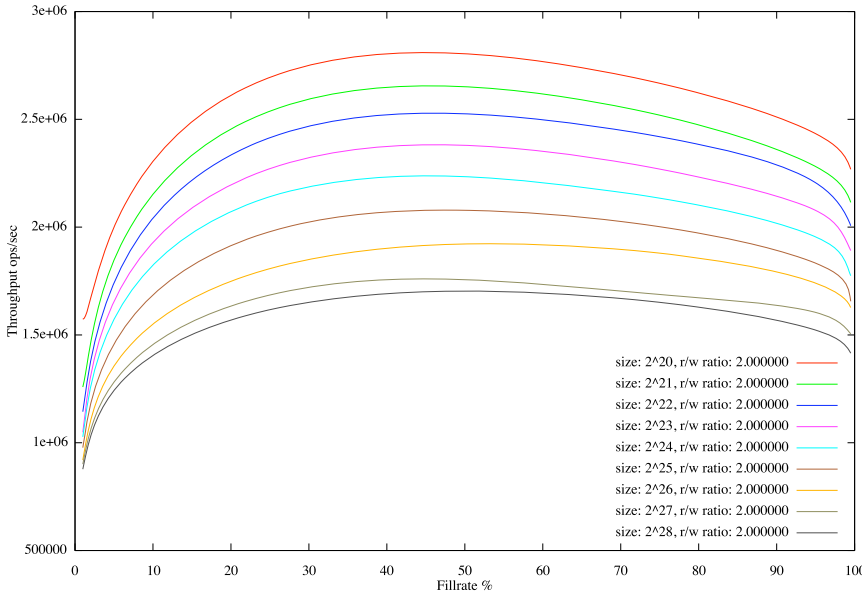
EXPERIMENTS

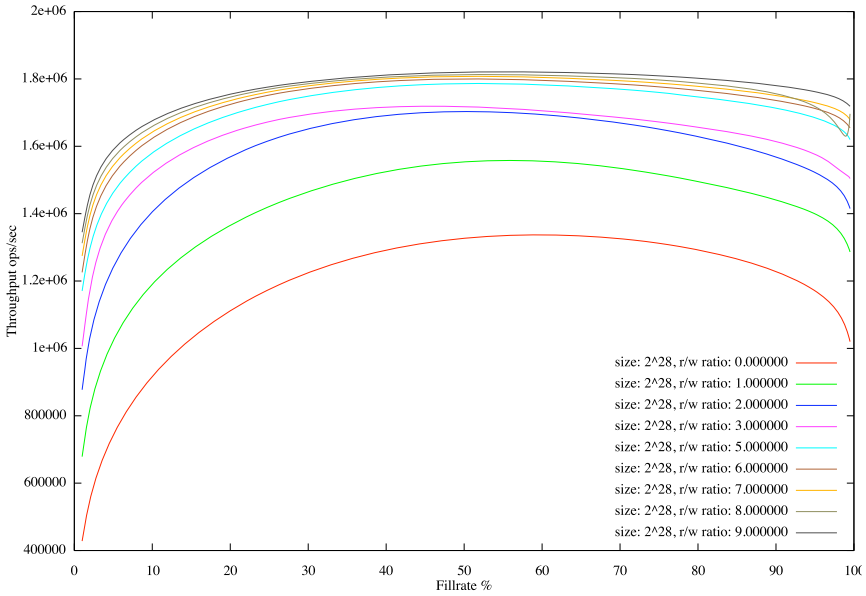




EXPERIMENTS







DISCUSSION / LIMITATIONS

- Statically sized states
- Reachability as a basis
- Slower sequentially
- Not 100% lock-free, but also not necessary

CONCLUSIONS

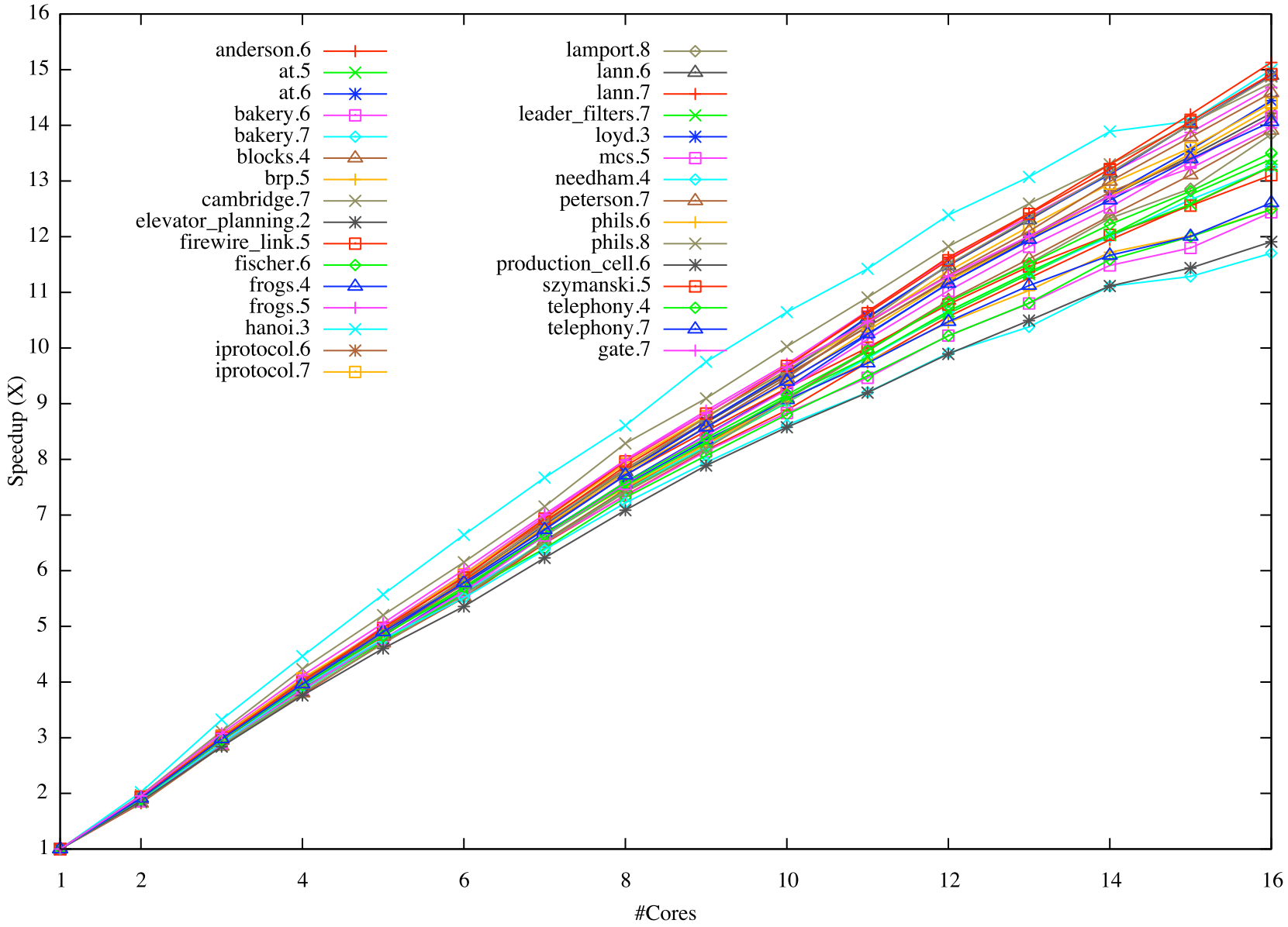
- Adapt applications to hardware (memory hierarchy)
- Centralized state storage scales better and is more flexible
- Scalable explicit exploration is a good starting point for future work on multi-core X. X ε {(weak) LTL model checking, symbolic exploration, space-efficient explicit exploration}

- Holzmann's conjectures:
  - works only for unoptimized sequential code
  - works only for small state vectors/long transition delays

  SPIN has many features though, but backwards compatibility seems wrong starting point for scalable multi-core algorithms
- HTTP://FMT.CS.UTWENTE.NL/TOOLS/LTSMIN/

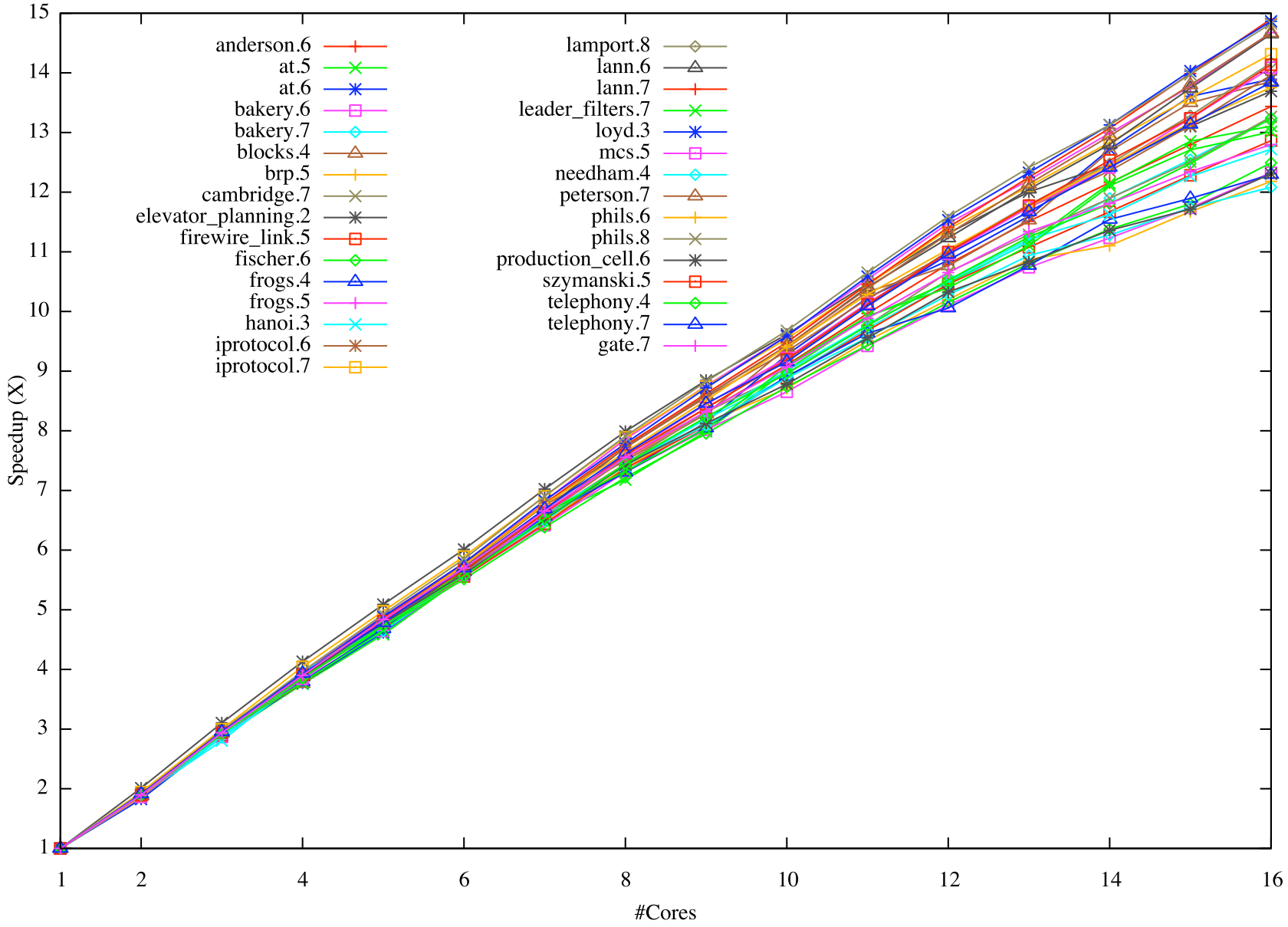
LTSMIN BFS SPEEDUPS

BASE CASE: LTSMIN BFS



LTSMIN DFS SPEEDUPS

BASE CASE: LTSMIN DFS



LTSMIN SPEEDUPS

BASE CASE: LTSMIN STATIC LOAD BALANCING