Logic Model Checking

Lecture Notes 17:18

Caltech CS 118

January-March 2006

Course Text:

The Spin Model Checker: Primer and Reference Manual

Addison-Wesley 2003, ISBN 0-321-22862-6, 608 pgs.

algorithmic techniques to reduce verification complexity (M*B*S)

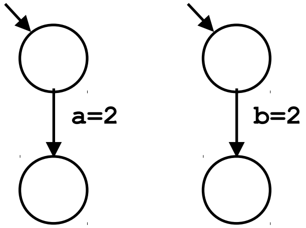
- to reduce B:
  - usually not an issue
  - for complex properties:
    - separate into smaller properties (it would be nice to have an algorithm for this)
    - try both ltl2ba -f and spin –f to see which algorithm produces the smaller automaton (neither is guaranteed to generate smaller automata than the other alas…)
- to reduce M:
  - partial order reduction (default in Spin)
  - abstraction (supported by Spin extension only)
  - symmetry reduction (supported by Spin extension only)
- to reduce S:
  - lossless compression (sharing, symbolic)
  - lossy compression (bitstate, supertrace)

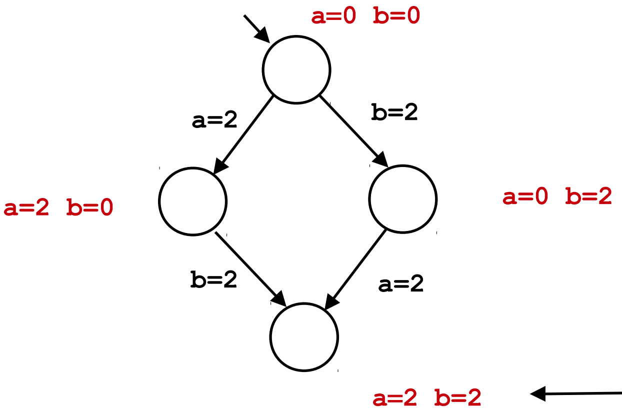
Partial-Order Reduction

partial order reduction

- full asynchronous interleaving of process actions is sometimes redundant

```
byte a, b;

active proctype A()
{
    a = 2; 0
}

active proctype B()
{
    b = 2; 0
}
```

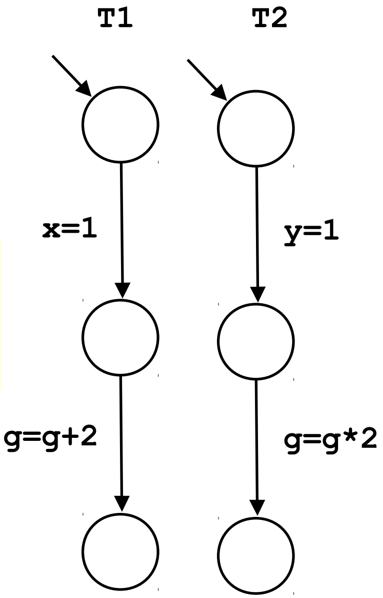




the final result is the same, no matter which path is followed

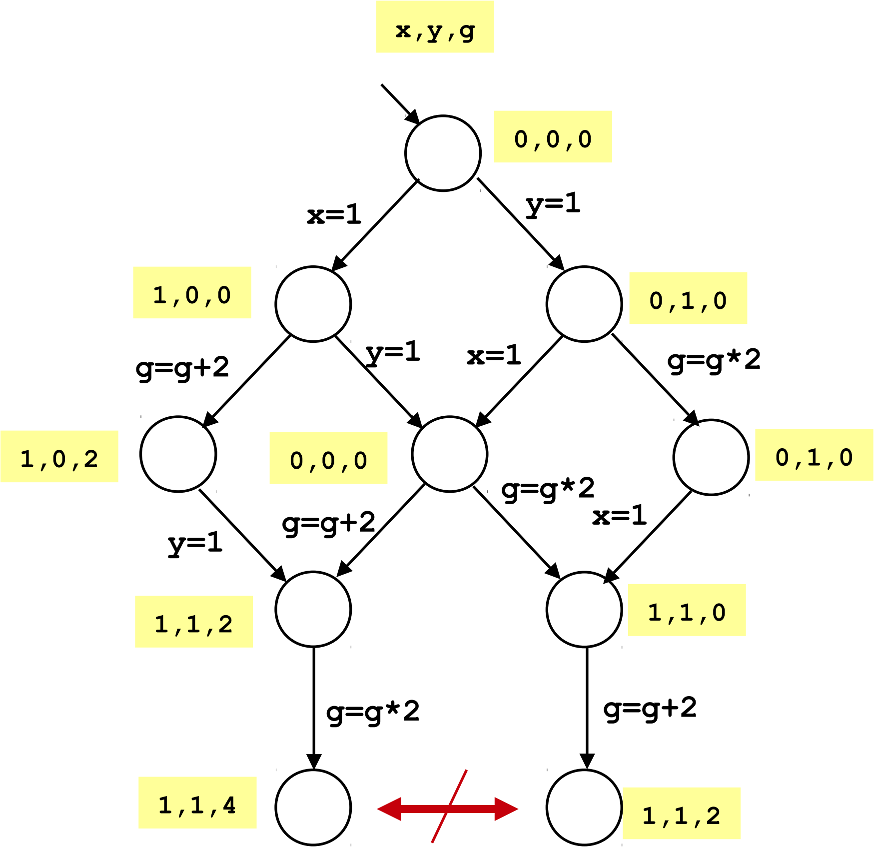
partial order reduction

a slightly larger example



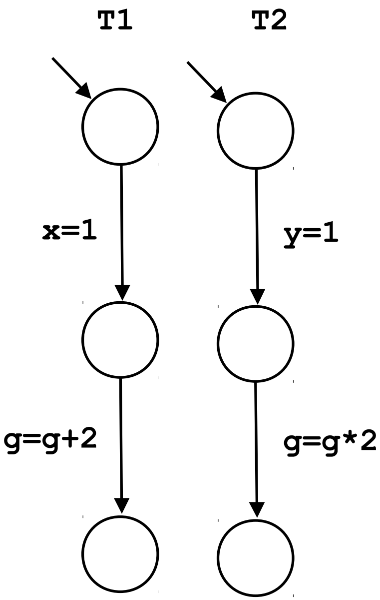
local variables:
x and y
global variable:
g

six runs:
`x=1;g=g+2;y=1;g=g*2`
`x=1;y=1;g=g+2;g=g*2`
`x=1;y=1;g=g*2;g=g+2`
`y=1;g=g*2;x=1;g=g+2`
`y=1;x=1;g=g*2;g=g+2`
`y=1;x=1;g=g+2;g=g*2`



```
only two operations share data:
          g=g+2 <-> g=g*2
all other combinations of operations
are data-independent, e.g. x=1 <-> g=g+2
```

data and control dependence



	x=1	y=1	g=g+2	g=g*2
x=1		`I`	`Control`	`I`
y=1	`I`		`I`	`Control`
g=g+2	`Control`	`I`		`Data`
g=g*2	`I`	`Control`	`Data`	

`I:` *Independent* operations
Control: control dependent operations
Data: data dependent operations

runs that differ only in the relative order of *independent* operations are equivalent

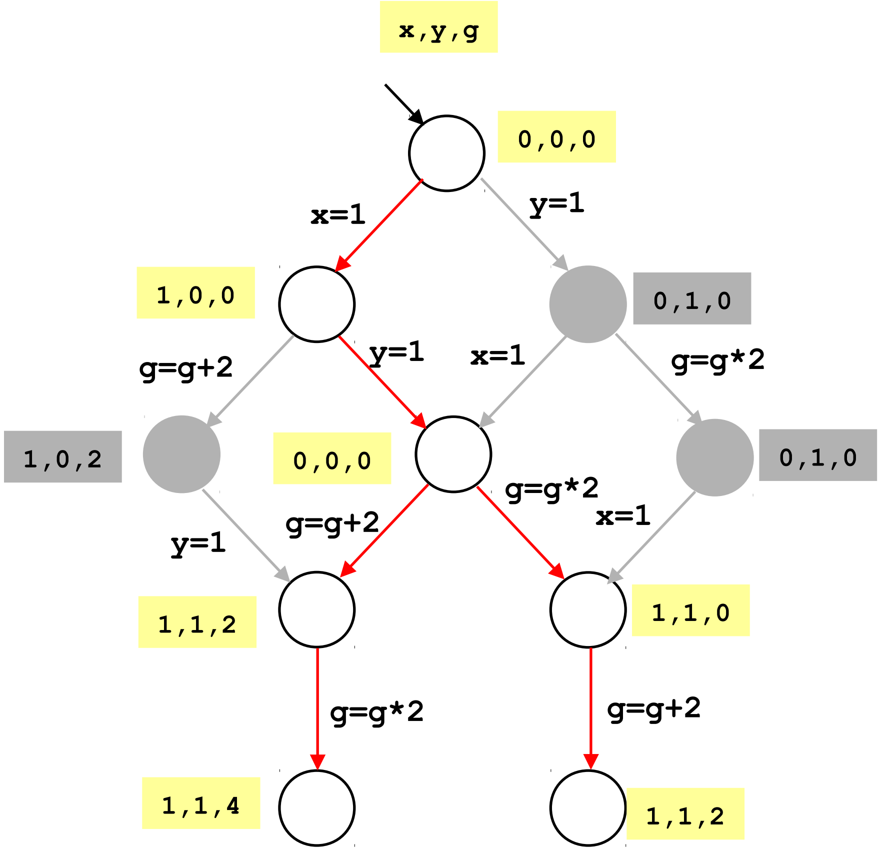
partial order reduction

independent pairs:
x=1 , y=1
x=1 , g=g*2
y=1 , g=g+2

2 groups of 3 equivalent runs each:

x=1;g=g+2;y=1;g=g*2
x=1;y=1;g=g+2;g=g*2
y=1;x=1;g=g+2;g=g*2

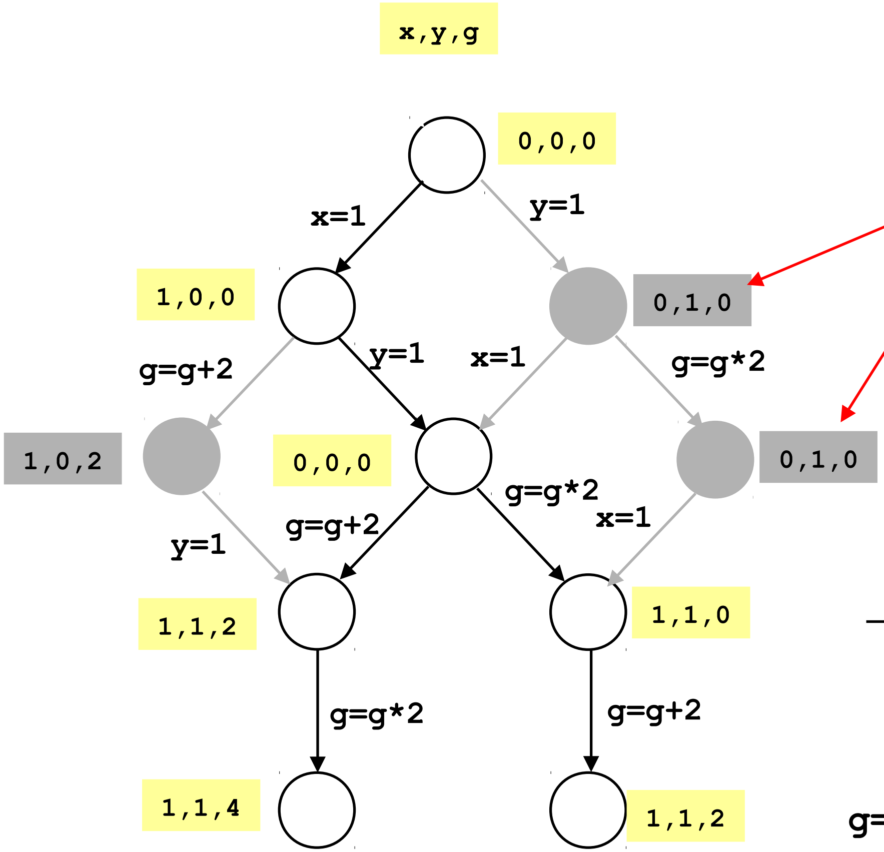
x=1;y=1;g=g*2;g=g+2
y=1;x=1;g=g*2;g=g+2
y=1;g=g*2;x=1;g=g+2



but what if we want to prove:
[] (x >= y)

reducing R from 10 to 7 states
(eliminating 3 states and 6 transitions)

visibility



[] (x >= y)

holds in the reduced graph, but *not* in the full graph

x and y are no longer independent

there is a 3rd class of dependence: property dependence (visibility)

	x=1	**y=1**	**g=g+2**	**g=g*2**
x=1		P	Control	I
y=1	P		I	Control
g=g+2	Control	I		Data
g=g*2	I	Control	Data	

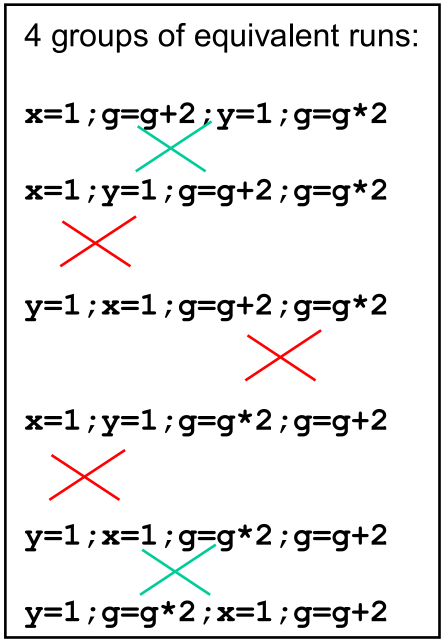
I: *Independent* operations
P: *Property* dependent (Visible)

visibility

	x=1	**y=1**	**g=g+2**	**g=g*2**
x=1		P	Control	I
y=1	P		I	Control
g=g+2	Control	I		Data
g=g*2	I	Control	Data	

`I`: *Independent* operations
P: *Property* dependent (Visible)

`independent pairs:`
~~x=1 , y=1~~
x=1 , g=g*2
y=1 , g=g+2



4 groups of equivalent runs:

x=1;g=g+2;y=1;g=g*2

x=1;y=1;g=g+2;g=g*2

y=1;x=1;g=g+2;g=g*2

x=1;y=1;g=g*2;g=g+2

y=1;x=1;g=g*2;g=g+2

y=1;g=g*2;x=1;g=g+2

slightly reduced reduction

4 groups of equivalent runs:

`x=1;g=g+2;y=1;g=g*2`

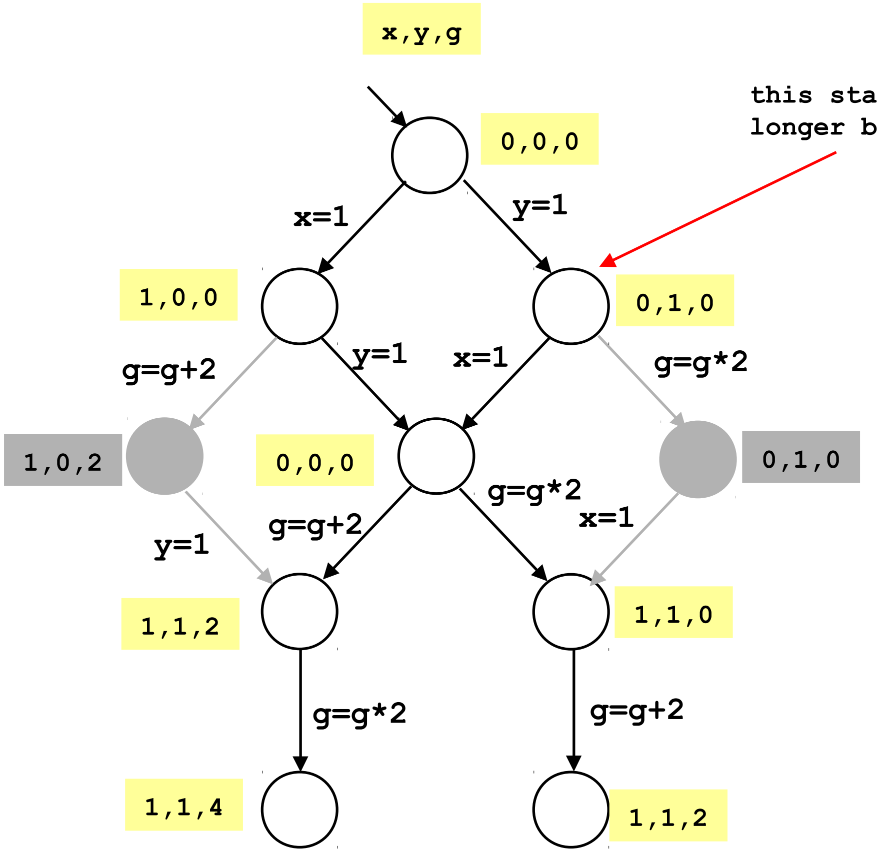
`x=1;y=1;g=g+2;g=g*2`

`y=1;x=1;g=g+2;g=g*2`

`x=1;y=1;g=g*2;g=g+2`

`y=1;x=1;g=g*2;g=g+2`

`y=1;g=g*2;x=1;g=g+2`



this state can no longer be eliminated

1 more state must be explored

partial order reduction

- two transitions are *independent* at state s if
  - both are enabled at s
  - the execution of neither can disable the other (no control dependence)
  - the combined effect of both transitions is independent of the relative order of execution (no data or property dependence)
- *strong* independence
  - two transitions are strongly independent if they are independent at every state where both are enabled
- safe transitions (this is a *static* property, that can be checked at compile time… to avoid runtime overhead for enforcing PO reduction)
  - a transition is *safe* if it is strongly independent from *all* other transitions in the system (Spin implementation)

reduction can be proven to preserve all safety and liveness properties (Peled, 1994)

the effect of even this *conservative* notion of independence can be an
exponential reduction
in the size of the reachable state space (M*B)
***without* measurable runtime overhead…**

Partial Order Reduction (ample set technique)

(C0) “if a state has at least one successor in the full state space, it has at least one successor in the reduced state space.”

(C1) “for all states s and for all paths in the full state space, starting at s, the following holds true: an action a that is dependent on an action b in ample(s) cannot be executed without a transition from ample(s) occurring first”. ***

(C2) “for all states s if s is not fully expanded, then every transition in ample(s) is invisible”;

(C3) “the reduced state graph may not contain a cycle in which an action a is enabled for some state s of the cycle so that a is not in the ample set of any state s' of the cycle”. ***

*** as hard as exploring the whole state space

C0-3 approximations in SPIN

1. Consider a simple set of candidates for ample(s), i.e. the set of transitions corresponding to each process.
   (ensures control-independency)

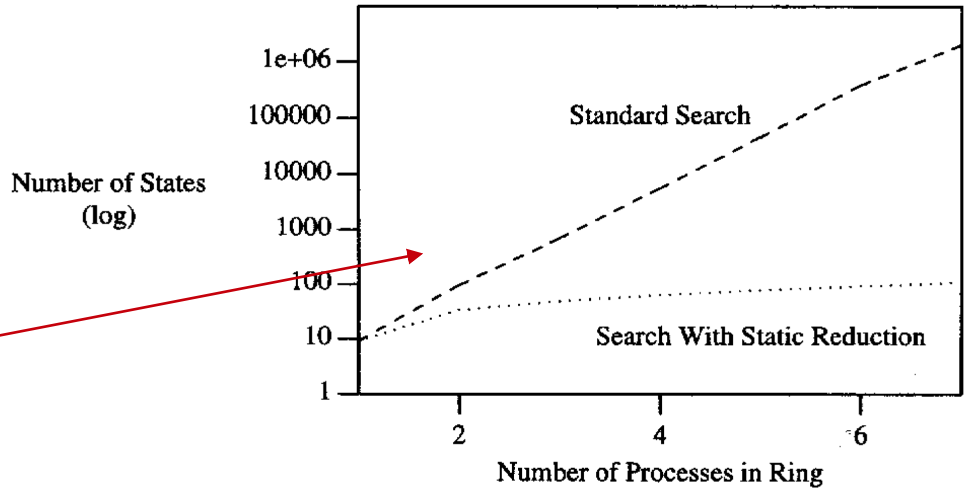
(C0)

(C1)

2. Discard empty ample sets (unless the state is a deadlock);
3. Consider ample sets with safe transitions only, i.e.
   (i) data independent from any other action *b* if:
      - *a* access local variables only;
      - *a* operates on a shared channel with exclusive access (only on process reads and only one process).

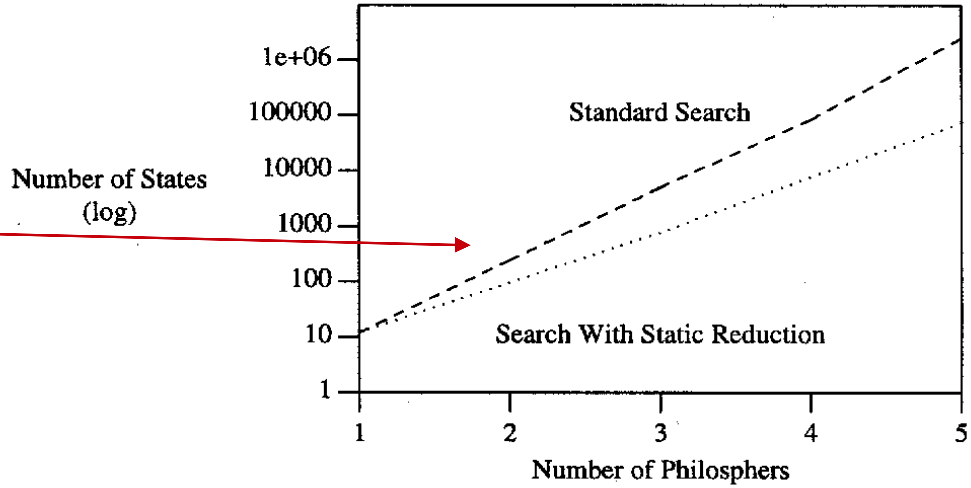
   (ii) property independent (i.e. invisible) (C2)
      - *a* modifies local variables only;
      - *a* modifies variables not used by the "never claim" (???)
4. If all successors of a state *s* are on the DFS stack (i.e. they all close a cycle) then expand all successors of *s*. (C3)

effect of partial order reduction



best case

Dining Philosphers (Dijkstra)



worst case

no partial order reduction

7 nodes

```
$ spin -a leader.pml
$ cc -DNOREDUCE pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 276 byte, depth reached 148, errors: 0
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 2.70635e+006 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
205.347 equivalent memory usage for states (...)
174.346 actual memory usage for states (compression: 84.90%)
        State-vector as stored = 233 byte + 8 byte overhead
1.049   memory used for hash table (-w18)
0.240   memory used for DFS stack (-m10000)
175.266 total actual memory usage

...

real    0m16.657s
user    0m0.015s
sys     0m0
```

default compression

175.3 Mbytes used
17 seconds

all states reached

effect of partial order reduction

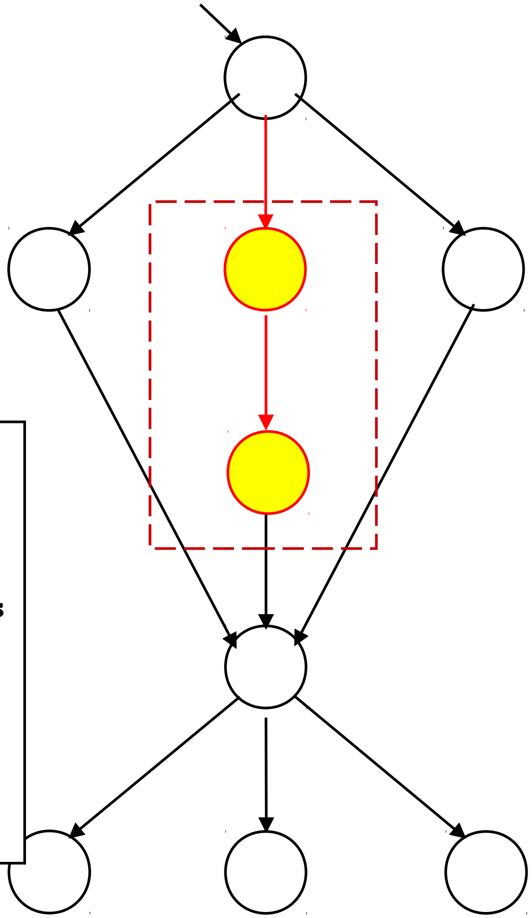
```
$ spin -a leader.pml
$ cc pan.c
$ time ./pan
(Spin Version 4.1.2 -- 4 February 2004)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 272 byte, depth reached 148, errors: 0
     133 states, stored
       0 states, matched
     133 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.573   memory usage (Mbyte)

unreached in proctype node
        line 53, state 28, "out!two,nr"
        (1 of 49 states)
unreached in proctype :init:
        (0 of 11 states)

real    0m0.076s
user    0m0.046s
sys     0m0.015s
$
```

```
175.3 Mbytes used
17 seconds

all states reached
```

```
1.5 Mbytes used
0.076 seconds

all relevant
states reached
```

statement merging (default spin reduction)

a form of partial order reduction



a sequence of unconditionally safe, non-blocking, transitions:

```
x = 1;
x = y+z;
```

predictably produces a non-interleaved run of states in the global graph

the intermediate states in such sub-graphs are redundant and can be omitted

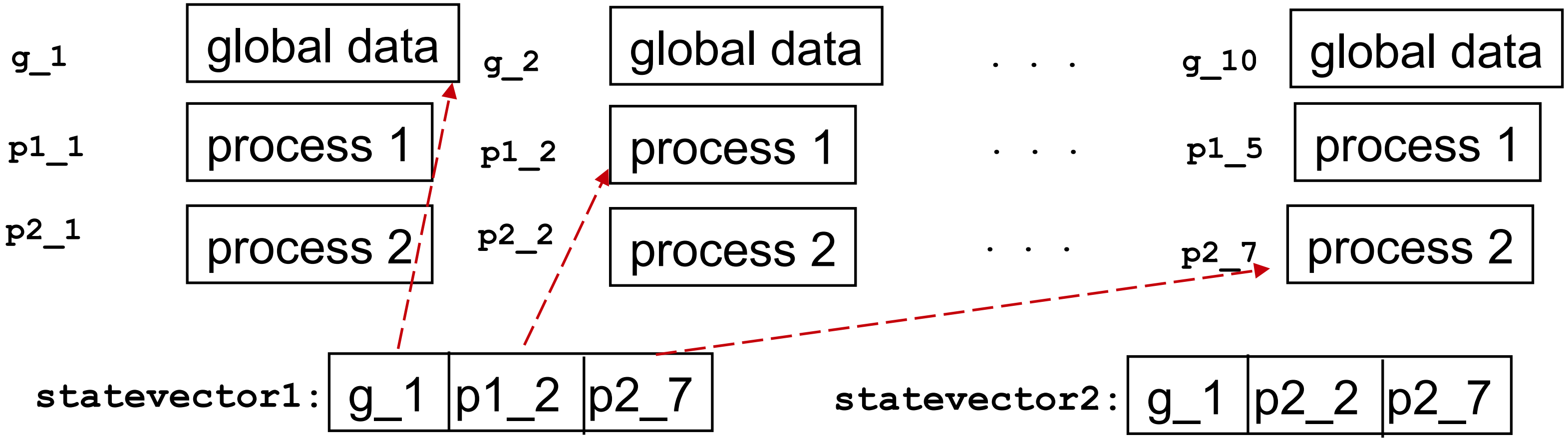
we can accomplish that effect by merging sequences of unconditionally safe transitions into a single transition (similar to d_step)

savings in memory and time

default in Spin
(can be disabled with spin -a -o3 …)

State Compression

state compression (-DCOLLAPSE)



`g_1` global data `g_2` global data . . . `g_10` global data

`p1_1` process 1 `p1_2` process 1 . . . `p1_5` process 1

`p2_1` process 2 `p2_2` process 2 . . . `p2_7` process 2

`statevector1:` g_1 | p1_2 | p2_7

`statevector2:` g_1 | p2_2 | p2_7

the state-vector is broken down into separate components:

- global data and message channels
- processes (one component for each active process)

each component is stored separately in a lookup table, and each component is given a unique *index-number*

only the *index numbers* are used to form the global state vector, which is stored in the statespace

basic idea: a small number of local component typically appear in many different combinations

effect of collapse compression

```
$ cc -DNOREDUCE -DCOLLAPSE pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Compression
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 276 byte, depth reached 148, errors: 0
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 3.23779e+006 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
208.239 equivalent memory usage for states (...)
23.547  actual memory usage for states (compression: 11.31%)
        State-vector as stored = 21 byte + 12 byte overhead
1.049   memory used for hash table (-w18)
0.240   memory used for DFS stack (-m10000)
24.738  total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 2765 129 2 ]
...

real    0m20.104s
user    0m0.015s
sys     0m0.015s.015s
```

```
175.3 Mbytes used
17 seconds

all states reached
```

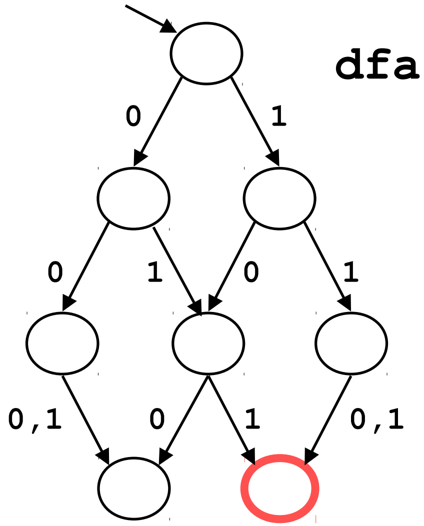
```
24.7 Mbytes used
20 seconds

all states reached
```

minimized dfa storage (-DMA)

instead of storing states explicitly in a hash-table, we can build a minimized deterministic finite automaton as a recognizer for states

example:

states = { 011, 101, 110, 111 }



`dfa recognizer`

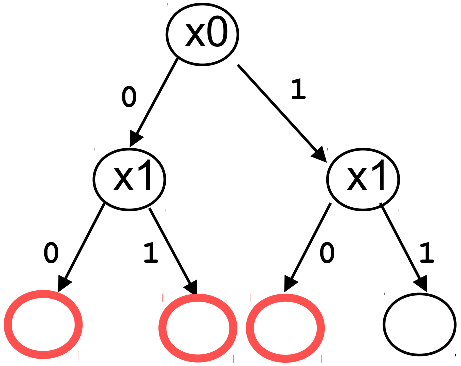
updating the DFA for a new state s takes O(|s|), but the constant factor is relatively large (compared to explicit storage)

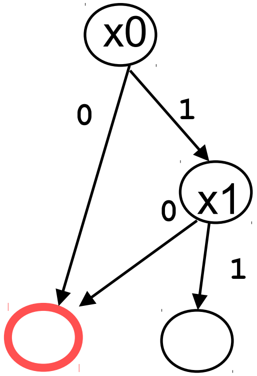
- can reduce memory use exponentially
- considerably more time consuming than explicit storage

short note on BDDs

Symbolic representation of states:

- Codify states as bit vectors x1,...,xn;
- A boolean formula over xi = v , v ∈ {0, 1} represents a set;
  E.g. $(\neg x0 \wedge \neg x1) \vee (x0 \wedge x1) \vee (x0 \wedge \neg x1)$
- Boolean formulae as Binary Decision Diagrams (BDDs)





- BDDs can efficiently represent states and compute transitions.
- (Bounded) Symbolic model checking via SAT

effect of minimized automaton storage

```
$ cc -DNOREDUCE -DMA=270 pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Graph Encoding (-DMA=270)

Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 276 byte, depth reached 148, errors: 0
MA stats: -DMA=234 is sufficient
Minimized Automaton:    161769 nodes and 397920 edges
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
202.455 equivalent memory usage for states (...)
7.235   actual memory usage for states (compression: 3.57%)
0.200   memory used for DFS stack (-m10000)
7.338   total actual memory usage
...
real    1m11.428s
user    0m0.015s
sys     0m0.015s
```

```
175.3 Mbytes used
17 seconds

all states reached
```

```
7.3 Mbytes used
71 seconds

all states reached
```

```
typical effect:
big reduction in Mem use
big increase in runtime
```

effect of using both
minimized automaton storage + collapse

```
$ cc -DNOREDUCE -DMA=21 -DCOLLAPSE pan.c
$ ./pan
(Spin Version 4.0.7 -- 1 August 2003)
        + Compression
        + Graph Encoding (-DMA=21)
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 276 byte, depth reached 148, errors: 0
Minimized Automaton:      5499 nodes and  25262 edges
  723053 states, stored
3.00211e+006 states, matched
3.72517e+006 transitions (= stored+matched)
      16 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

Stats on memory usage (in Megabytes):
208.239 equivalent memory usage for states (...)
0.892   actual memory usage for states (compression: 0.43%)
1.049   memory used for hash table (-w18)
0.200   memory used for DFS stack (-m10000)
2.068   total actual memory usage

nr of templates: [ globals chans procs ]
collapse counts: [ 2765 129 2 ]
...
real    0m44.214s
user    0m0.015s
sys     0m0.015s
```

```
175.3 Mbytes used
17 seconds

all states reached
```

```
2 Mbytes used
44 seconds

all states reached
```

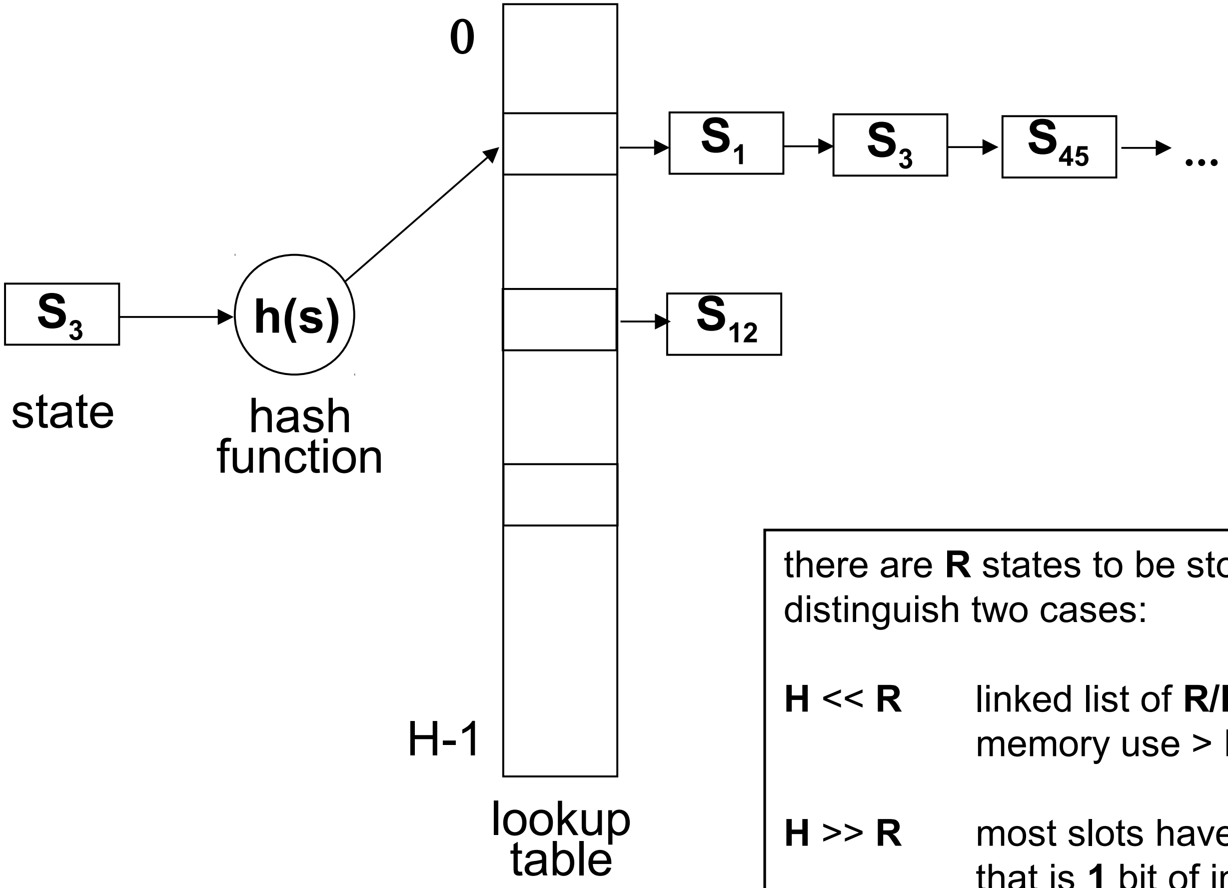
```
not always as effective
as it is in this case
```

bitstate hashing: lossy storage

(the *supertrace* algorithm from 1987)

- instead of explicitly storing all reachable states we will now store only a few bits per state
- in an attempt to optimize search coverage and minimize memory use and runtime
  - assume R states, S bytes per state, M bytes of memory available; the intended area of application for bitstate hashing is when we cannot do a standard search, i.e.:
    - R*S >> M
  - we can accept a small probability of incompleteness, provided that we miss *significantly fewer* states than would be missed in a normal run that exhausts available memory
    - reaching far more states than M/S
    - but, no *guarantee* that we will always reach *all* R states

state storage: hash-tables



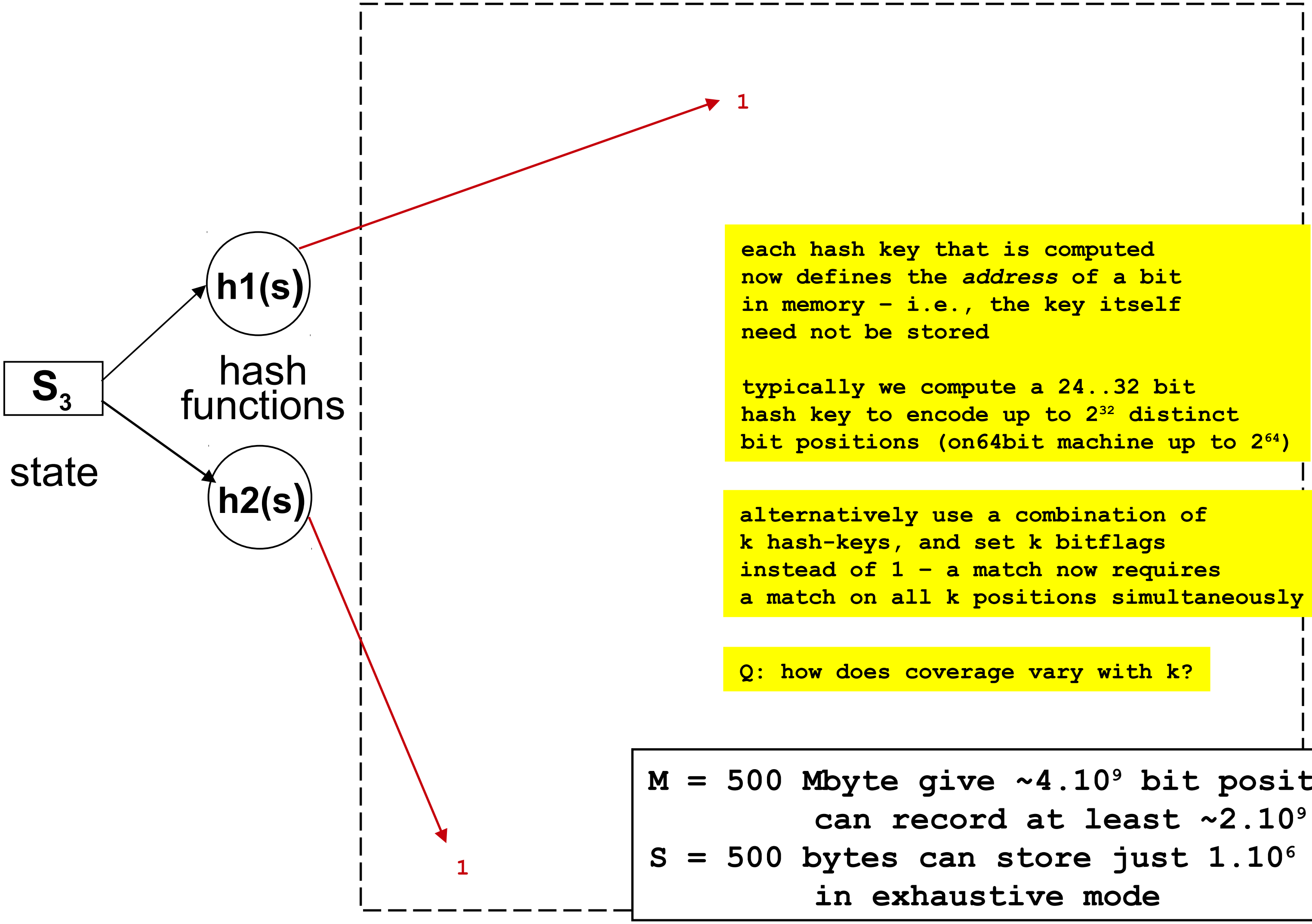
there are **R** states to be stored;
distinguish two cases:

H << **R**	linked list of **R/H** states per slot memory use > **R.S** bytes
H >> **R**	most slots have 0 or 1 state that is **1** bit of information per slot effective memory use **R** bits

Robert Morris [CACM1968]

- in the case where H >> R there is *no need to store* the hash-key...
- the possibility of a hash-collision now becomes remote
  - "*no-one to this author's knowledge has ever implemented this idea, and if anyone has, he might well not admit it.*"
- trading increased memory use for increased accuracy:
  - instead of 1 hash-function, use k>1 independent hash-functions
  - "store" each state k times
  - a hash-collision now requires k matches
  - Spin originally used *2* CRC polynomials to compute the hashes
  - current version uses *3* by default, user can choose any other number

the bitstate array



each hash key that is computed now defines the *address* of a bit in memory – i.e., the key itself need not be stored

typically we compute a 24..32 bit hash key to encode up to 2^{32} distinct bit positions (on64bit machine up to 2^{64})

alternatively use a combination of k hash-keys, and set k bitflags instead of 1 – a match now requires a match on all k positions simultaneously

Q: how does coverage vary with k?

M = 500 Mbyte give ~4.10^9 bit positions, can record at least ~2.10^9 states

S = 500 bytes can store just 1.10^6 states in exhaustive mode

effect of collisions:

causes *possible* incompleteness of search

but, accuracy of error reports is always preserved

- If a hash collision happens, the target state is assumed to have been visited, while in fact it was not
- This means that the target state is missed
  - if target is an error state, that error may be missed
- Are all successors of the missed state also missed?
  - not necessarily, in an asynchronous process system there are typically many different paths that lead to the same state: the same set of states can be reached in many ways, so if one of the paths is blocked, another path will likely still find the state and its successors
- What about errors that are found
  - they will always be accurate and indistinguishable from errors reported in an exhaustive search – the path on the *stack* identifies the execution sequence leading to the error as before

Bloom filters (Burton Bloom, 1970)

- k independent hash-functions – setting k bit-positions
- initially the hash-array has all zero bits: assume m bits.
- after r states have been stored, the probability of a specific bit being zero is:

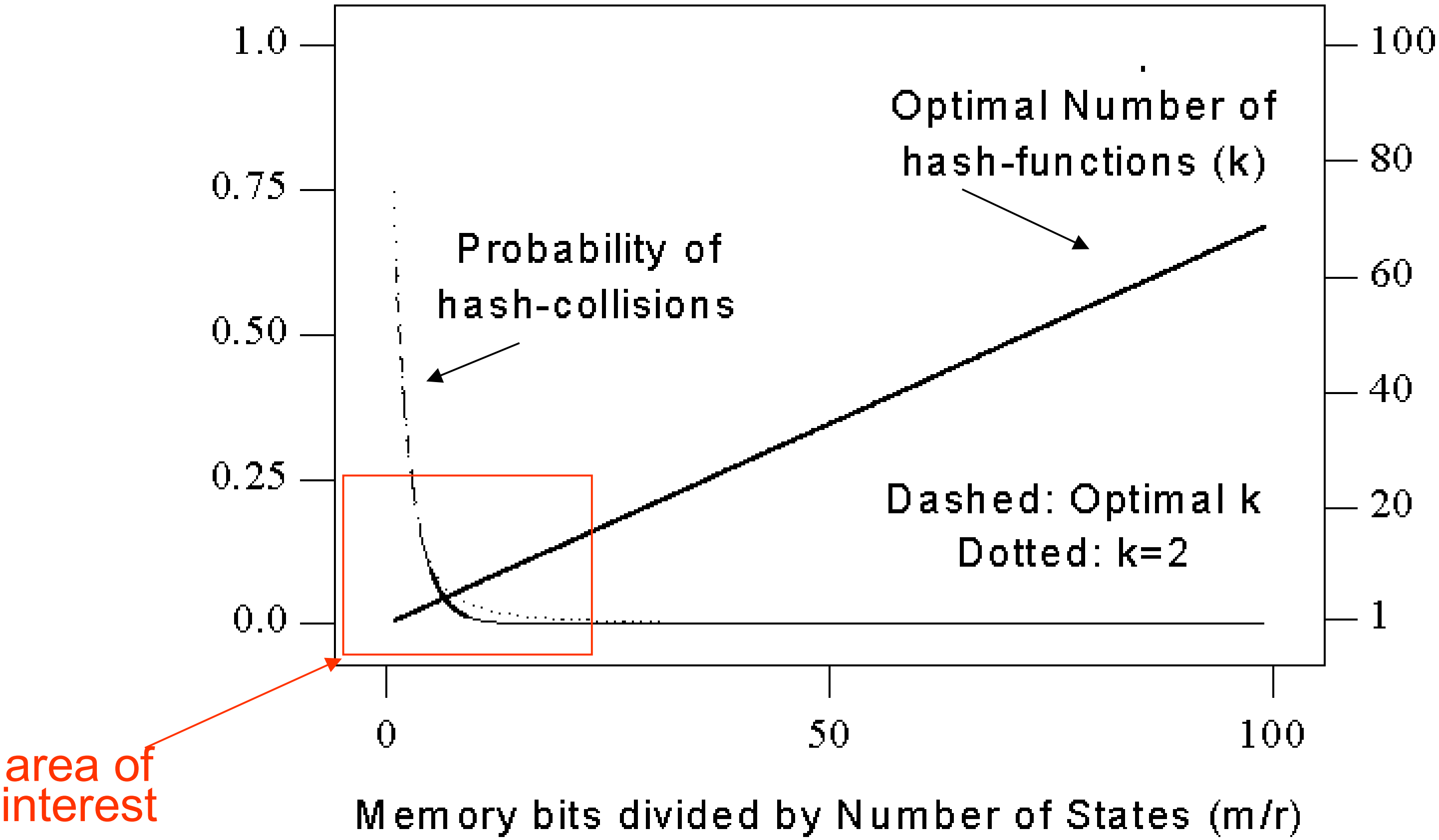
$$r \times (1 - \frac{1}{m})^k$$

the probability of a hash-collision on the (r+1)th entry:

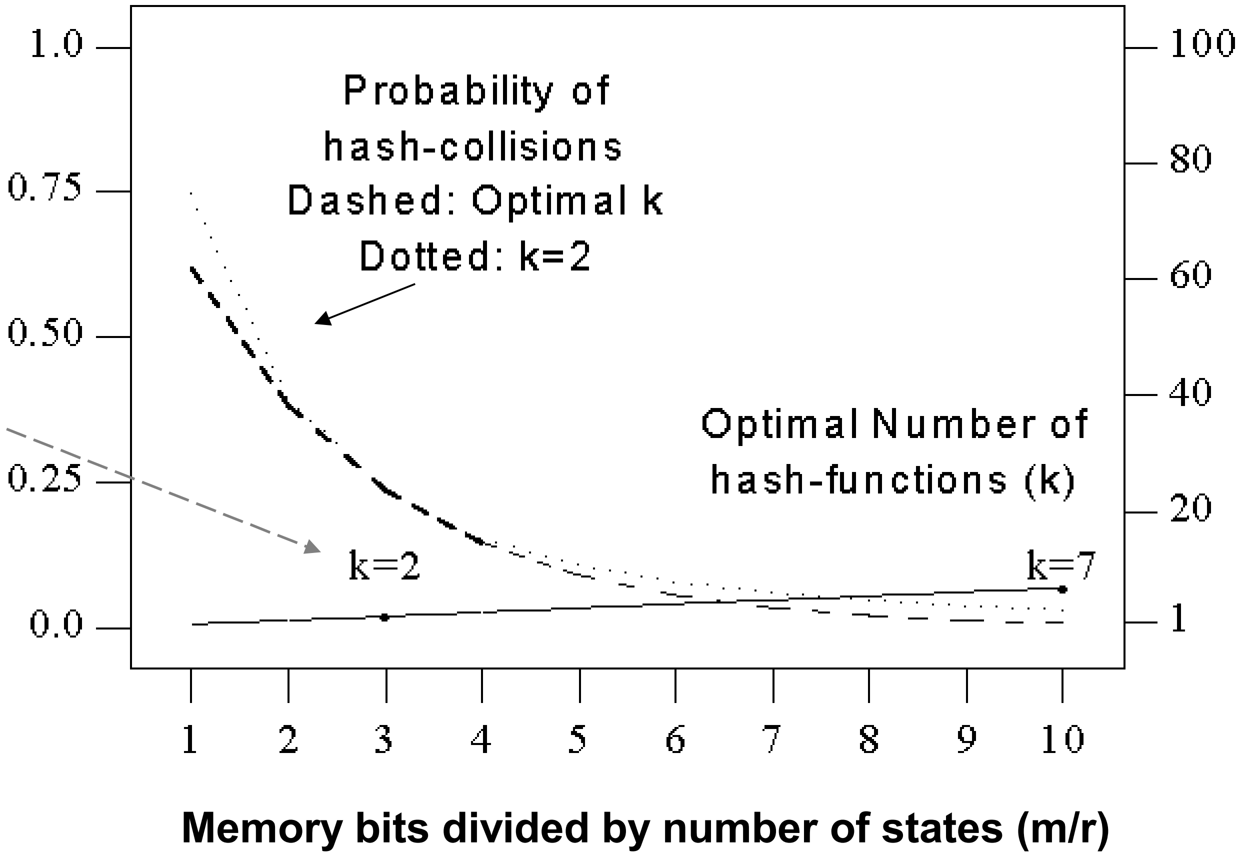
$$1 - (r \times (1 - \frac{1}{m})^k)^k \approx (1 - e^{-k \cdot r/m})^k$$

the right-hand side is minimized for k = ln 2 x m/r

probability of hash-collisions
optimal number of hash-functions



probability of hash-collisions optimal number of hash-functions



Spin's default: k = 3

bitstate

```
$ spin -a leader.pml
$ cc -DNOREDUCE -DBITSTATE -o pan pan.c
$ time ./pan
(Spin Version 4.0.7 -- 1 August 2003)

Bit statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 276 byte, depth reached 148, errors: 0
  700457 states, stored
2.9073e+006 states, matched
3.60775e+006 transitions (= stored+matched)
      16 atomic steps
hash factor: 5.98795 (best coverage if >100)
(max size 2^22 states)

Stats on memory usage (in Megabytes):
198.930 equivalent memory usage for states (...)
0.524   memory used for hash array (-w22)
2.097   memory used for bit stack
0.240   memory used for DFS stack (-m10000)
3.066   total actual memory usage
...
real    0m28.550s
user    0m0.015s
sys     0m0.015s
```

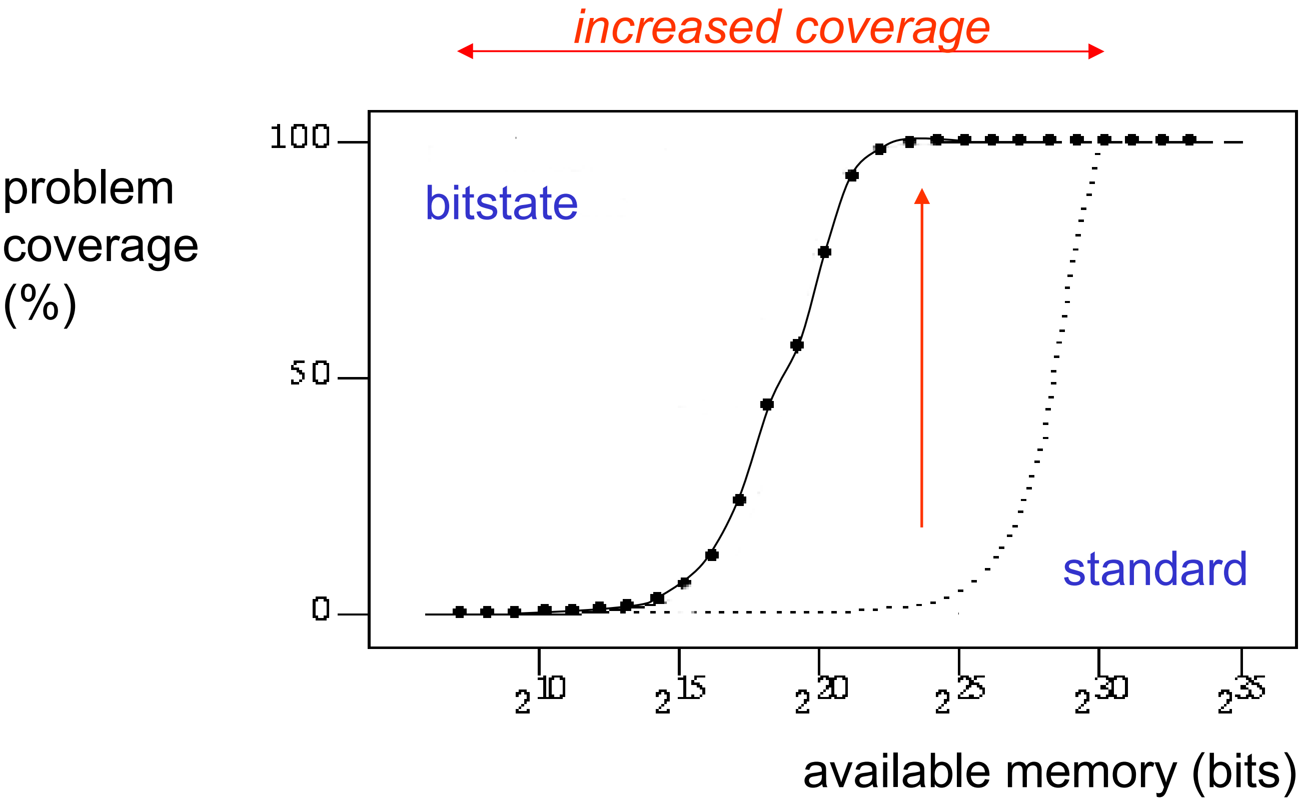
175.3 Mbytes used
17 seconds

all states reached

3 Mbytes used
28 seconds

96.7% of all states reached

effect of bitstate hashing increased search coverage



(Data: a Commercial Data Transfer Protocol)

accuracy vs speed

- by shrinking the available memory arena, we increase speed and reduce coverage
- the effect of the hash functions is that the search space is pruned randomly, so we can use bitstate hashing to perform a fast random pre-scan of a search space
  - with *user-selectable* accuracy and speed
- this makes it possible to do *iterative* search refinement
  - start with a search arena of 64k bits, run verifier, if an error is found stop, if not: double the search arena and repeat
  - until either an error is found or an exhaustive search was completed

options options

partial order reduction	no downside, default mode
statement merging	no downside, default mode
-DCOLLAPSE	good compression; small time penalty
-DMA	superb compression; large time penalty
-DBITSTATE	superb compression; chance of loss; fast