On Limits

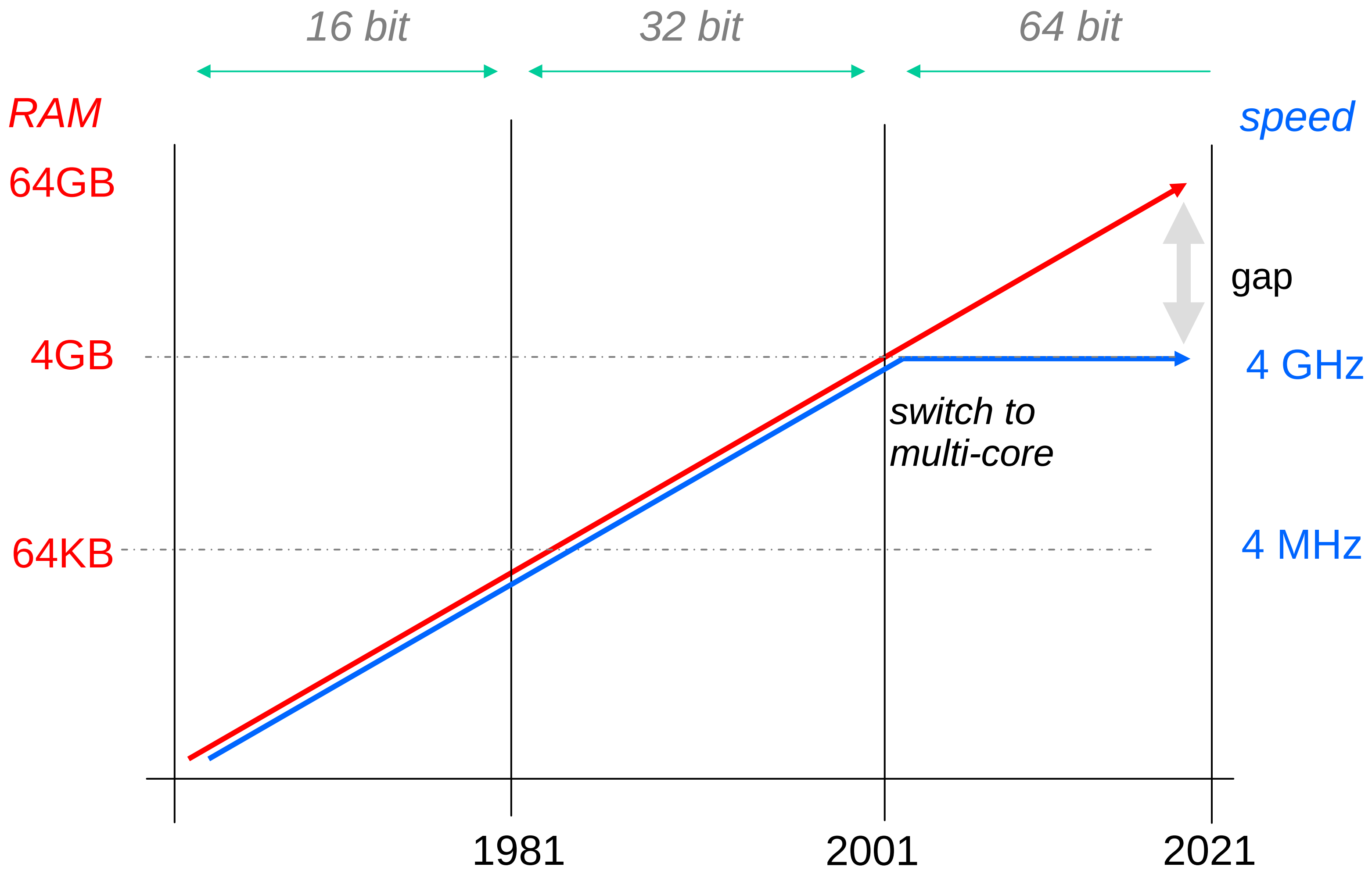
Gerard J. Holzmann

Jet Propulsion Laboratory

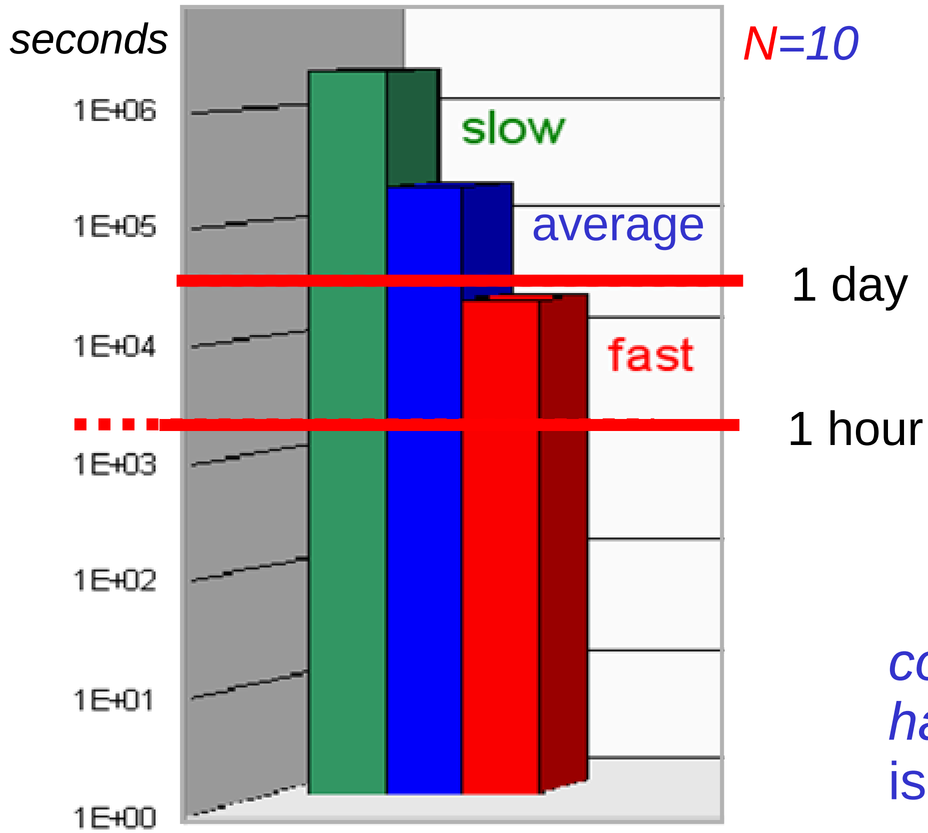
California Institute of Technology

speed and memory trends

(we will soon have very large amounts of memory and relatively slow processors)



the time needed to fill *N* GB of RAM



conclusion:
having more memory
is not always useful

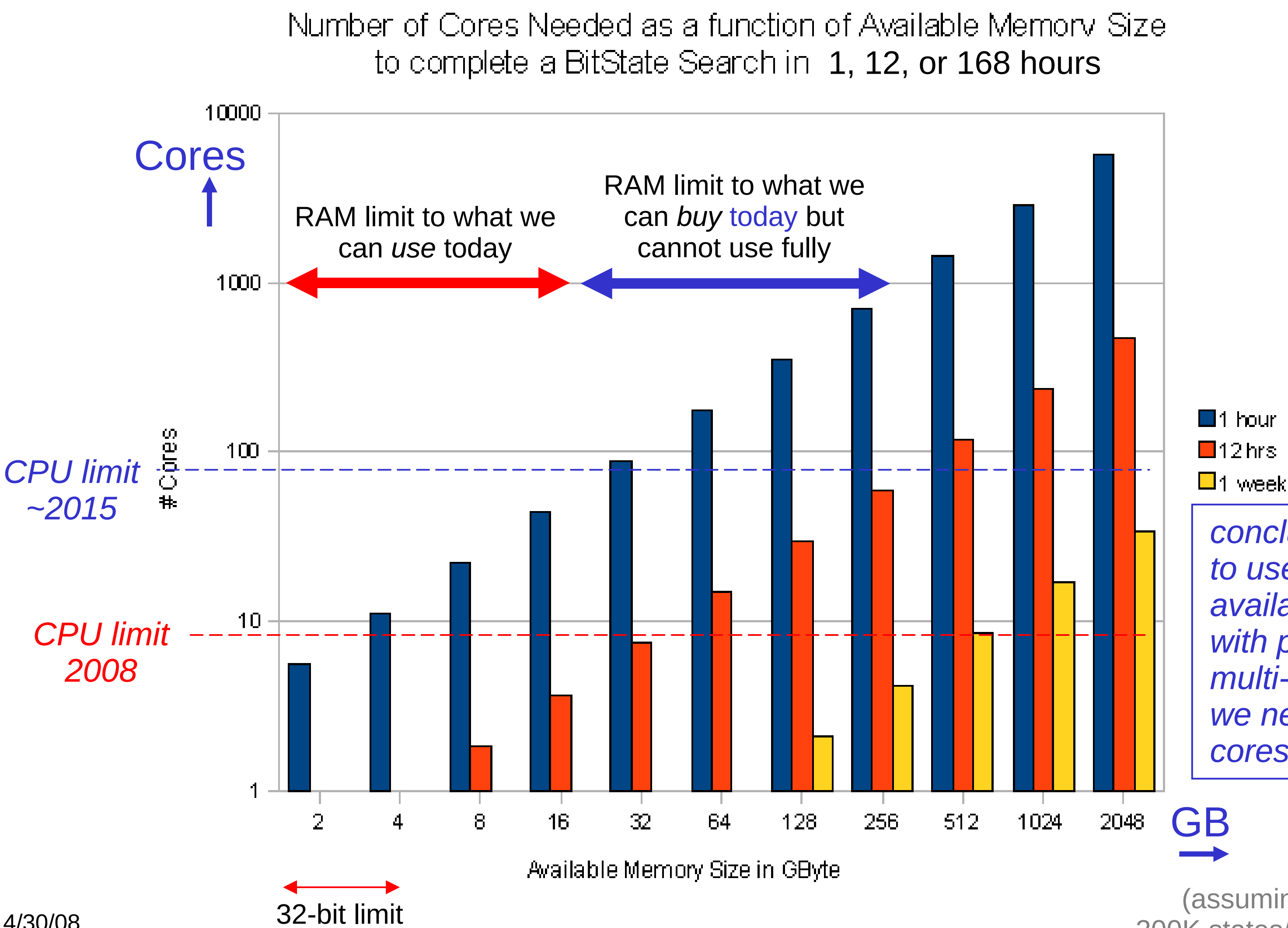
[Spin in bitstate mode]
storing a relatively large number of system states
into memory at a rate of 10^4 to 10^6 states/second

what are the limits?

- at a fixed clock-speed, there is a limit to the largest problem size we can handle in 1 hour (day / week)
  - no matter how much memory we have (RAM or disk)
  - even a machine with “infinite memory” but “finite speed” will impose such limits

- we can increase speed by using multi-core algorithms
  - but do 10^n CPUs always get a 10^n x speedup?
  - it will depend on the CPU architecture (NUMA/UMA)
  - do we know what the CPU architecture will be for large multi-core machines (think 1,000 CPUs and up)?
- isn’t there an easier way?
  - can’t we find a way to use N x as many CPUs, and get a result that is always “N x better” (by some definition of “better”)

at fixed speed how many CPUs does it take to fill up *N* GB of RAM in 1 hour?

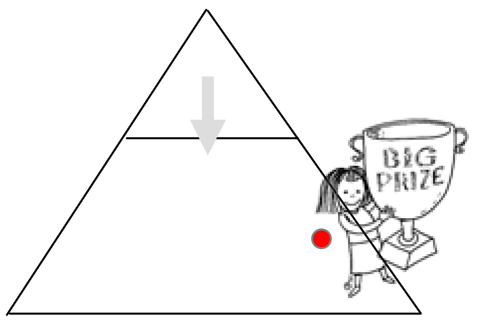


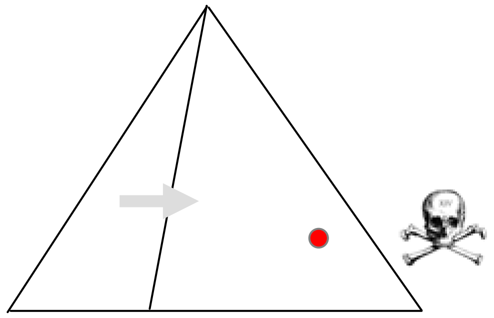
conclusion:
to use all GB of RAM available in year Y, with perfect multi-core scaling we need the nr of CPU cores from year Y+7

(assuming: ~200K states/second bitstate search)

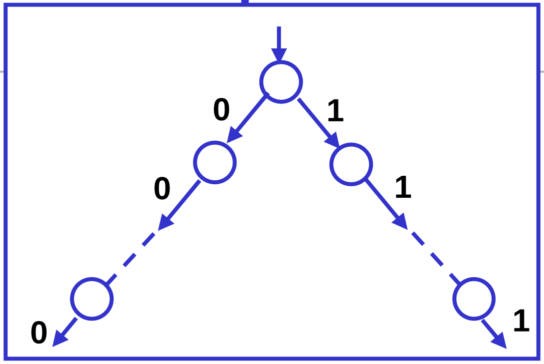
the infinitely large problem and the infinitely large machine

- there will always be problems that require more *time* to verify than we are willing (or able) to wait for
  - how do we best use finite time to handle large problems?
- example of an “infinitely large problem:” a Spin Fleet Architecture model from Ivan Sutherland & students (courtesy Sanjit Seshia)
  - known error state is just beyond reach of a breadth-first search (and symbolic methods) – error is too deep
  - error is on “wrong” side of the DFS tree
  - a bitstate search either fills up memory or exhausts the available time before the error state is reached
  - how do we maximize our chances of finding errors like this?





measurement:
define a simple, large search problem




```
byte pos = 0;
int val = 0;
int flag = 1;

active proctype word()
{    /* generate all 32-bit values */
end: do
     :: d_step { pos < 32 -> /* leave bit 0 */ flag = flag << 1; pos++ }
     :: d_step { pos < 32 -> val = val | flag; flag = flag << 1; pos++ }
     od
}

never {/* check if some user-defined value N can be matched */
    do
    :: assert(val != N)
    od
}
```

2^{32} reachable states, 24 byte per state

100 GB to store the full state space

what if we only have 64 MB to do the search?

0.06 % of what is needed

a sample search query

- 2^{32} reachable states, 24 bytes per state
  - 100 GB to store the full state space
  - 64 MB available (0.06 % of 100 GB)

- question:
  - seed *100* randomly chosen numbers
  - how many of these numbers can be found (matched)?
    - using different search techniques

- one obvious candidate: bitstate hashing with depth-first search
  - assume 0.5 byte per state on average: $2^{32} \times 0.5 \sim 2$ GB
  - 64MB (2^{26}) is now 3% (1/32) of what is needed to represent all states
  - should find matches for ~ 3 of the 100 numbers

bitstate dfs –w29

2^{29} bits = 2^{26} bytes = 64 MB

```
$ spin -DN=-1 -a word.pml
$ cc -O2 -DSAFETY -DBITSTATE -o pan pan.c
$ ./pan -w29
...
1.4849945e+08 states, stored  (3.46% of all 2^32 states)
...
hash factor: 3.61531 (best if > 100.)
bits set per state: 3 (-k3)
...
pan: elapsed time 127 seconds
$
```

this search does not find a match for the target number -1
if we repeat this 100x for each of the randomly chosen numbers
we should expect 3 or 4 matches

checking 100 numbers

```
$ > out
$ for r in `cat ../numbers`
$ do
   spin -DN=$r -a word.pml
   cc -O2 -DSAFETY -DBITSTATE pan.c
   ./pan –w29 >> out
done
$ grep “assertion violated” out | sort –u | wc -l
8
```

we were “entitled” to 3 or 4 matches, and we got 8 (i.e., we were lucky)

numbers matched:
234, -3136, 3435, 19440, 6985, 12435, 4915, 27246
(note: 52 of our targets are negative numbers, we matched only 1 in this subset)

using iterative search refinement [HS99]

(using 128KB, 256KB, … 64 MB)

```
$ > out
$ for w in 20 21 22 23 24 25 26 27 28 29
do
   for r in `cat ../numbers`
   do
        spin -DN=$r -a word.pml
        cc -O2 -DSAFETY -DBITSTATE pan.c
    ./pan –w$w >> out
   done
done
$ grep “assertion violated” out | sort –u | wc -l
15
```

-w	dfs
20	1
21	1
22	2
23	2
24	2
25	3
26	6
27	8
28	11
29	15

we increased the number of matches from 8 to 15
can we do still better?

adding search diversification

- dfs: standard depth-first search (the default)
- dfs_r: reverse order in which non-deterministic choices within a process are explored
  - using compiler directive –D_TREVERSE (Spin 5.1.5).
- r_dfs: use search randomization on the order in which non-deterministic choices within a process are explored
  - using compiler directive –DRANDOMIZE (Spin 4.2.2)

    randomly selects a starting point in the transition list, and checks transitions for executability in round-robin order from that point

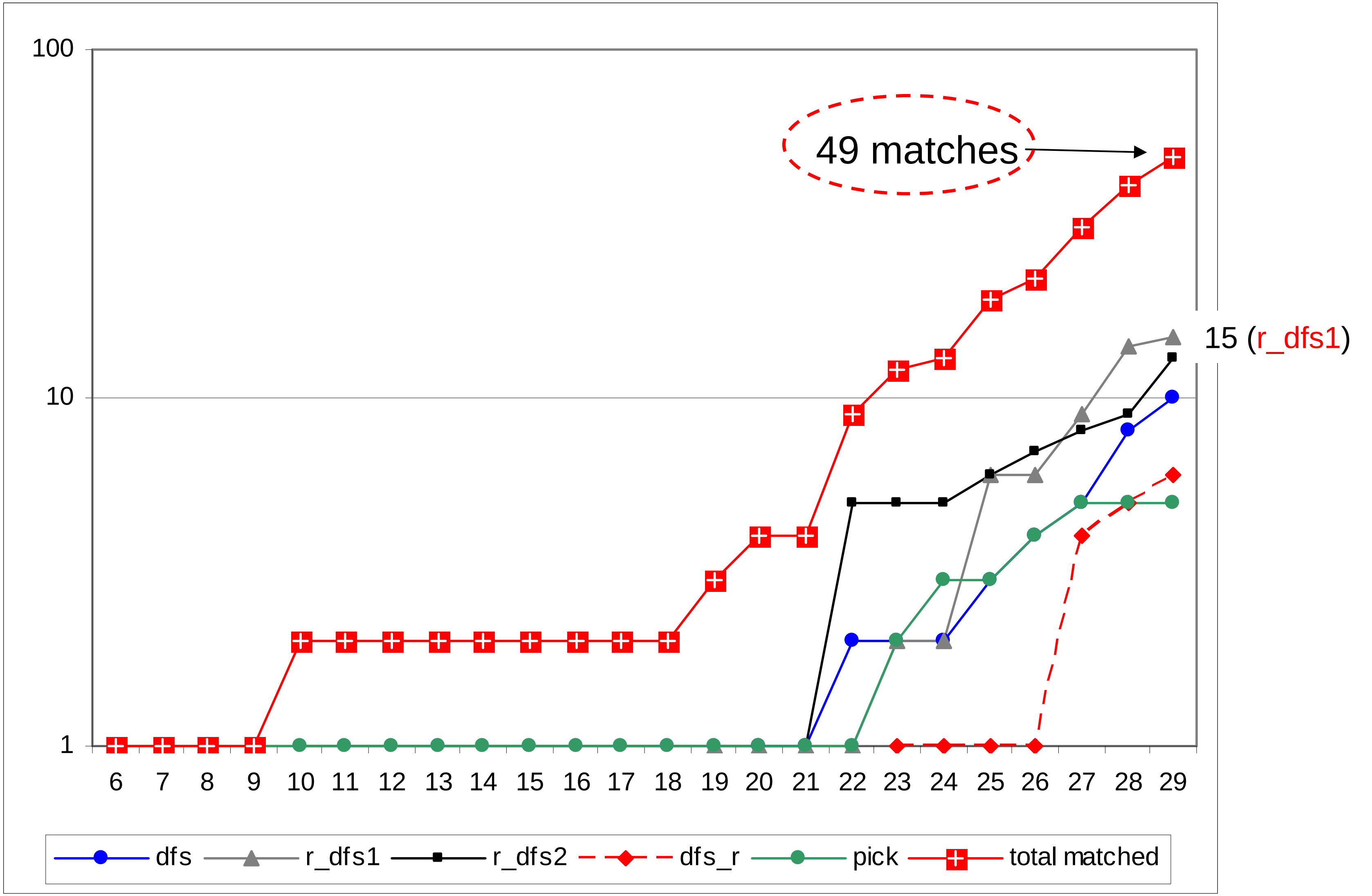
    use different seeds to create multiple variants (r_dfs1, r_dfs2)
- pick: use embedded C code to define a user-controlled selection method to permute the transitions in a list of non-deterministic choices within a process

pick: user-defined randomization

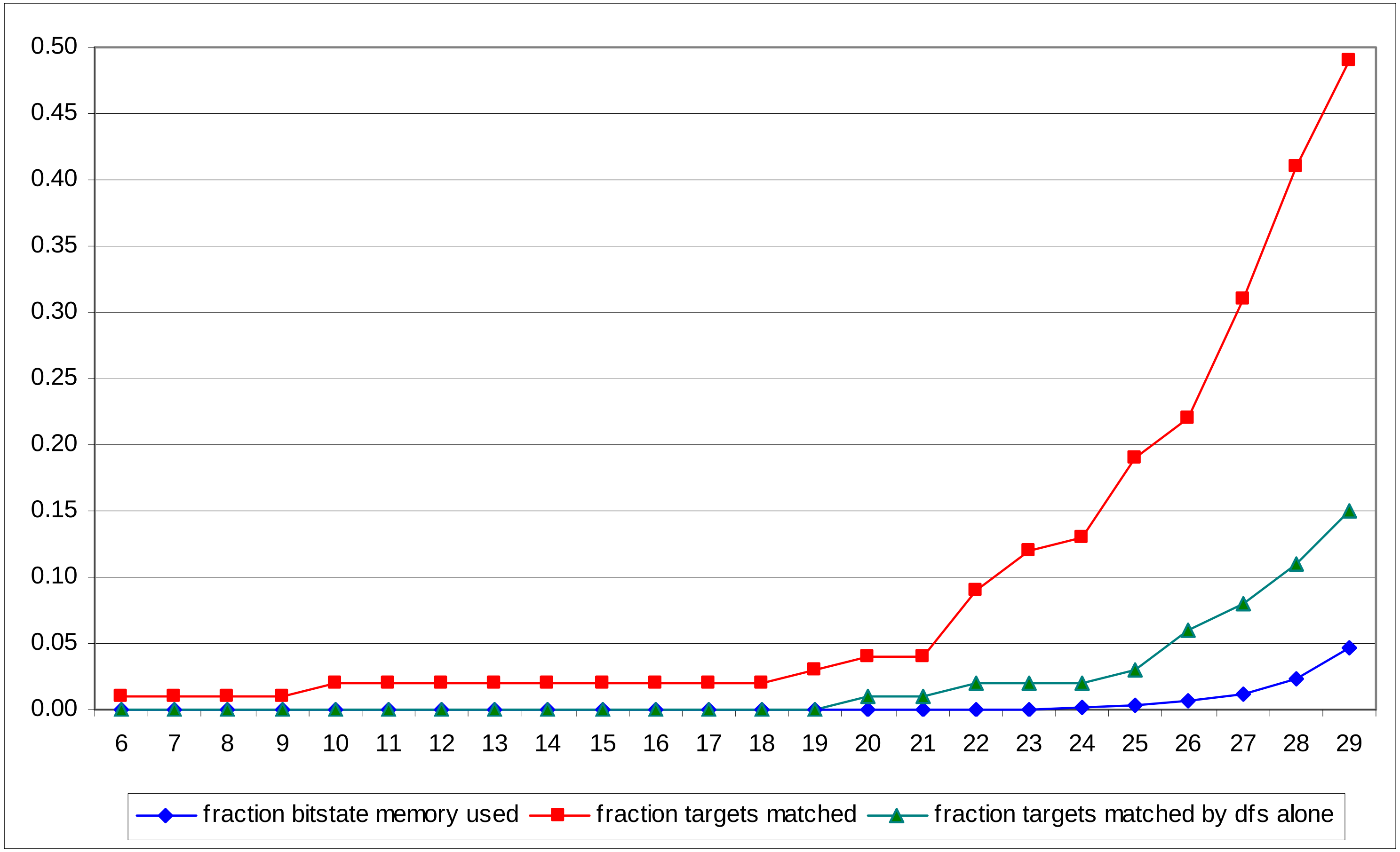
(courtesy of rajeev joshi & alex groce)

```
c_decl {
        \#define MAX_CHOICES  32   /* max nr of choices in calls to "pick" */

        int choices[MAX_CHOICES];
        int last_seed = 3;
};

c_track "choices"     "sizeof(int) * MAX_CHOICES"   "UnMatched";
c_track "&last_seed"  "sizeof(int)"                 "UnMatched";

inline pick(v, min, max) {
        tmp = max-min+1 ;
        c_code {
                int i, j, t;     /* temporary C vars */

                srandom(last_seed) ;
                for (i = 0; i < now.tmp; i++)
                {       choices[i] = i;
                }
                for (i = 0; i < now.tmp-1; i++)
                {       j = (random() % (now.tmp - i));
                        t = choices[i];
                        choices[i] = choices[i+j];
                        choices[i+j] = t;
                }
                now.tmp = 0;
        };
        /* randomize search order each time a node is revisited */
        do      /* cover all choices */
        :: d_step { tmp < max-min -> tmp++ }
        :: d_step {
                v = min + c_expr { choices[now.tmp] };
                c_code { last_seed += now.tmp; now.tmp = 0; }
           }; break
        od
}

int n, x, y, tmp;

active proctype main()
{
        do
        :: n < 3 -> n++;
                pick(x, 1, 3);
                pick(y, 7, 9);
                printf("n=%d, x = %d, y = %d\n", n, x, y)
        :: else ->
                break
        od
}
```

iterative search refinement + search diversification: nr matches increases to 49



fraction of memory used
compared with fraction of targets matched



(the memory reference is minimal amount of memory needed for full bitstate storage)

swarm

$ swarm –F config.lib –c6 > script
swarm: 456 runs, avg time per cpu 3599.2 sec
$ sh ./script

sample configuration file:

```
# ranges
w       20      32      # min and max -w parameter
d       100     10000   # min and max search depth
k       2       5       # min and max nr of hash functions

# limits
cpus            2               # nr available cpus
memory          513MB           # max memory to be used; recognizes MB,GB
time            1h              # max time to be used; h=hr, m=min, s=sec
vector          500     # bytes per state, used for estimates
speed           250000          # states per second processed
file            model.pml       # file with spin model

# compilation options (each line defines a search mode)
-DBITSTATE                      # standard dfs
-DBITSTATE -DREVERSE# reversed process ordering
-DBITSTATE -DT_REVERSE          # reversed transition ordering
-DBITSTATE -DRANDOMIZE=123              # randomized transition ordering
-DBITSTATE -DRANDOMIZE=173573   # ditto, with different seed
-DBITSTATE -DT_REVERSE -DREVERSE        # combination
 -DBITSTATE -DT_REVERSE -DRANDOMIZE # combination

# runtime options
-n
```

swarm verification of some large models

Verification Model	State vector size	System states reached in standard bitstate dfs (-w29)	Time for bitstate dfs (in minutes using 1 cpu)	Number of swarm jobs (1 hour limit 6 cpus)
EO1	2736	320.9M	43	86
Fleet	1440	280.5M	58	228
DEOS	576	22.3M	2	456
Gurdag	964	86.2M	17	231
CP	344	165.7M	18	451
DS1	3426	208.6M	159	100
NVDS	180	151.2M	6	516
NVFS	212	139.5M	45	265

performance

Verification Model	Number of Control States			% of Control States Reached	
	Total	Unreached			
		standard dfs	dfs + swarm	standard dfs	dfs + swarm
EO1	3915	3597	656	8	83
Fleet	171	34	16	80	91
DEOS	2917	1989	84	32	97
Gurdag	1461	853	0	41	100
CP	1848	1332	0	28	100
DS1	133	54	0	59	100
NVDS	296	95	0	68	100
NVFS	3623	1529	0	58	100

synopsis

- there is a growing performance gap
  - memory sizes continue to grow
  - but cpu speed no longer does (for now)
  - the standard approaches to handling large problem sizes have stopped working
  - we have to get smarter about defining incomplete searches in very large state spaces
- the best use of currently available computational resources (and *human time*)
  - may be to switch to the use of embarrassingly parallel methods, in combination with search diversification