trends in cpu memory and clock-speed

memory: 16 bit, 32 bit, 64 bit

clock-speed: 4 MHz, 4 GHz

gap: 4 GHz x 256 CPUs

To leverage the new trend, we should try to find ways to exploit massive parallelism.

(twenty year intervals)
time to fill $N$ GB of RAM

Storing a relatively large number of system states into memory at a rate of $10^4$ to $10^6$ states/second

- 1 day
- 1 hour

$N=10$

More memory is no longer always more useful if only because life itself is finite…
some observations

• 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

• in some cases 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)?
**NUMA**

Altix C-Brick
4 CPUs
(2x dual-cpu)
2 NUMA links

up to 16 GB per C-brick

0.4---3.2 Gbps

16 C-Bricks = 64 CPUs
max 16x16=256 GB

on this architecture, we can expect to see performance changes at 2, 4, 16, and 64 CPU boundaries

64 CPUs NUMA interconnect (non-uniform memory access)

measurement on the SGI Altix

each bar records the runtime of 1 of N processes

2 GB per process (left) or 2 GB shared memory (right)

with shared memory, some processes are always faster, likely cpus near the data
but, once the faster processes terminate, the slow processes don’t migrate to the data…

all memory references local
(note, runtimes measured tend to match in multiples of 2 or 4)

using any number of processes ≥ 8 leads to a major performance hit

(uncertainty in measurements: we have no control over how the scheduler assigns processes to cpus)
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?

• an 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?
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
assume we have only 64 MB to do the search
0.06 % of what is needed to store everything
finding needles in haystacks

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

• a search problem:
  – randomly pick 100 32-bit numbers
  – how many of these numbers can we find (match) with different search techniques?
  – the odds of finding any of the numbers with a standard exhaustive search are not very good…

• a first candidate: bitstate hashing
  – consumes ~0.5 byte per state on average: $2^{32} \times 0.5 \sim 2$ GB
  – 64MB ($2^{26}$) is 1/32 of what is needed to store all bit-states
  – should find matches for ~3% of the 100 numbers
bitstate dfs –w29

$ 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 150 seconds

this search did not find a match for the target number -1

but, if we repeat the search for each of the 100 numbers we can expect maybe 3 matches
let’s try it

```bash
$ > out
$ for r in `cat ../numbers` # 100 separate runs
$ do
   spin -DN=$r -a word.pml
   cc -O2 -DSAFETY -DBITSTATE -o pan pan.c
   ./pan -w29 >> out
done
$ grep "assertion violated" out | sort -u | wc -l
```

2

two numbers were matched: -1904, 30754

can we do better?
but why do 100 runs, when we can do 1

```c
active proctype word()
{
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 {
  do
:: d_step { pos == 32 ->
    if
    :: (val == -29786)
    || (val == -8747)
    || (val == 234)
    || ...
    || (val == -9934) ->
      c_code { printf("assertion violated %d\n", val); }
:: else
    fi }
:: else
  od
}
```

runtime goes from 100 x 150 seconds (> 4 hours) down to 180 seconds
(but note that it removes potential parallelism)
we’ll use this run as a reference

```bash
$ spin -a word_100.pml
$ cc -O2 -DSAFETY -DBITSTATE -o pan pan.c
$ ./pan -w29 -k3 -h0
```

We can try adding search diversity to see if we can increase problem coverage:

1. change hash-polynomials (default is –h0, can use –h1..32)
2. change the number of hash-functions (default is –k3, can use any k)
3. change the size of the hash-array (up to 64MB: can use -w1..29)
4. change search algorithm… (we’ll come back to this)

Each variation defines an independent run, that can be executed completely in parallel – without any sharing.

Does any of this really buy us anything?

the challenge: increase coverage above 2-3%, without increasing memory or time…
changing hash-polynomials

```bash
$ > out
$ for h in 0 5 11 17 # possible choices: 0..32
do
   ./pan -w29 -k3 -h$h >> out
done
$ grep "assertion violated" out | sort -u | wc -l
```

this *tripled* the number of matches by varying 1 parameter.

we defined 4 independent runs.

what if we also vary *k* and *w*?

varying *w* is an older technique, called "iterative search refinement" in [HS99]
creating 160 runs
by varying 3 parameters

```bash
$ > out
$ for w in 20 21 22 23 24 25 26 27 28 29 # 10 bitstate sizes
do
  for k in 1 2 3 4 # 1 to 4 hash-functions
do
    for h in 0 5 11 17 # 4 hash-polynomials
do
      ./pan -w$w -k$k -h$h >> out
done
done
$ grep "assertion violated" out | sort -u | wc -l
14
```

we now locate 14% of our 100 search targets
all 160 runs are independent and can be executed in parallel – most runs are very fast
we can also vary the search algorithm
three simple methods:

1. standard depth-first search our reference
2. reverse the order for exploring transitions within a process
   • compile pan.c with –D_T_REVERSE
3. add search randomization on the transition selections within a process
   • compile pan.c with –DRANDOMIZE=N
   • in our case, we have just 2 transitions, but the choice between them is made 32 times in each of the 4 billion possible executions
   • can use different seeds to create any number of variants

   each search variant can be expected to perform roughly the same, but each should hit different targets, so that all variants combined can outperform any one variant used separately.
we can use this to define a large nr of runs
e.g., 30 x 160 = 4,800 parallel runs

```bash
for x in dfs rdfs 433 33461 593 139 `seq 101 3 170``
done
case "$x" in
dfs) cc -O2 -DSAFETY -DBITSTATE -o pan pan.c ;;
rdfs) cc -O2 -DSAFETY -DBITSTATE -DT_REVERSE -o pan pan.c ;;
*) cc -O2 -DSAFETY -DBITSTATE -DRANDOMIZE=$x -o pan pan.c ;;
esac
... [the earlier script,
with 160 variations
for each algorithm]
done
```

the complete set can still be run in 180 s
on a compute grid / cloud / mesh / cluster

keep a few hundred cpus busy…
(something we to be able to do to
to solve very large problem sizes
in logic model checking very fast)
Increasing Problem Coverage with Search Diversity

- Each iteration is a set of 160 runs.
- Individual and cumulative number of matches.
- 98 matches.
- Each of 30 iterations is a set of 160 runs.
- No run uses more than 64 MB: 0.06% of the 100GB needed.
- No run takes more than 180 seconds.
- No run finds more than 2 targets.
- All runs are independent, and can be executed in parallel.
there are more ways to diversify the search...

4. use embedded C code to define a user-controlled selection method to permute the transitions selections

5. reverse the order in which processes themselves are interleaved
   • compile pan.c with –DREVERSE (not helpful here, since we have just 1 process)

6. breadth-first search
   • compile with –DBFS (not helpful here, since all targets are at the same level)

7. multi-core search
   • compile with –DNCORE=N (not explored here)

8. different types of bounds
   • Bounded context switching (as proposed by Shaz Qadeer -- to be implemented)
   • Depth-Bounded Search (varying -m…)
   • Bounded Storage (e.g., 2,3,4-byte hash-compact variations)
the swarm tool: a new preprocessor for Spin

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

sample swarm 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 128 # nr available cpus
memory 64MB # 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 word_100.pml # the 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 real-world verification models

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

<table>
<thead>
<tr>
<th>Verification Model</th>
<th>Number of Control States</th>
<th>% of Control States Reached</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Total</td>
<td>Unreached</td>
</tr>
<tr>
<td>EO1</td>
<td>3915</td>
<td>3597</td>
</tr>
<tr>
<td>Fleet</td>
<td>171</td>
<td>34</td>
</tr>
<tr>
<td>DEOS</td>
<td>2917</td>
<td>1989</td>
</tr>
<tr>
<td>Gurdag</td>
<td>1461</td>
<td>853</td>
</tr>
<tr>
<td>CP</td>
<td>1848</td>
<td>1332</td>
</tr>
<tr>
<td>DS1</td>
<td>133</td>
<td>54</td>
</tr>
<tr>
<td>NVDS</td>
<td>296</td>
<td>95</td>
</tr>
<tr>
<td>NVFS</td>
<td>3623</td>
<td>1529</td>
</tr>
</tbody>
</table>
synopsis

• there is a growing performance gap
  – memory continues to grow
  – but cpu speed no longer does (for now)
  – the standard approaches to handling large
    problem sizes has stopped working
  – we have to get smarter about defining
    incomplete searches in very large state
    spaces

• swarm leverages
  – search diversification and simple,
    embarrassingly parallel execution
http://spinroot.com/swarm/