Multi-Core model checking

- Multi-Core has become the dominant trend
  - No More Moore
- To leverage this change:
  - Extend logic model checking algorithms
    - Not targeting special purpose hardware (clusters), but desktops
    - This means: multi-core & shared memory
    - Should be possible to get automatic scaling of performance with a growing number of cores
  - Support all verification & storage modes in Spin
    - Safety & Liveness (including LTL, up to \( \omega \)-regular properties)
    - Bitstate hashing, hashcompact, exhaustive storage, etc.
    - Partial order reduction should work the same
- A potential hurdle: distributed model checking algorithms
  - Have been studied for many years
    - Mostly targeting compute clusters – few target shared memory
    - Mostly restricting to Safety properties – no good solutions for Liveness
    - Results often incomparable – few benchmarks
**what can we hope to achieve**

**design tradeoffs**

<table>
<thead>
<tr>
<th>CPU Performance:</th>
<th>relevant factors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Checker Performance:</td>
<td></td>
</tr>
<tr>
<td>Multi-Core PC (Shared memory)</td>
<td>CPU-cluster (Distributed memory)</td>
</tr>
<tr>
<td>Fast mc: 10</td>
<td>Fast mc: 10</td>
</tr>
<tr>
<td>Slow mc: 50</td>
<td>Slow mc: 50</td>
</tr>
<tr>
<td>RAM to RAM (memcpy) 3 µsec</td>
<td>relative time to transfer a state to another CPU 1 200</td>
</tr>
<tr>
<td>RAM to network port 600 µsec</td>
<td>relative time to generate a new state &amp; check if it is previously visited</td>
</tr>
</tbody>
</table>

**what can we hope to achieve**

speedup with increasing amounts of decoupling

**hypothesis 1:**
unoptimized implementations will benefit more than optimized implementations of model checkers

**hypothesis 2:**
multi-core platforms realize performance gains more easily than cluster computer systems (a 10-core PC may realize better performance than a 100-cpu cluster)
basic framework
multi-core model checking, with shared memory

- At selected points in the search, a CPU can hand off a state to another CPU, by adding it to the target’s work queue.
  - Using algorithms for locking access to shared data, and for distributed termination detection (verifiable with standard Spin.)
  - The state space arena can be shared (default) or non-shared (optional).

- A Spin extension for dual-core
  - ~900 lines of new code, supporting all relevant verification modes including LTL, compatible with partial order reduction – no increase in computational complexity.
  - The dual-core algorithm for safety properties scales to N-core systems – verification of liveness properties so far benefits only dual-core (i.e., it is an open problem to do liveness verification on N-cores without increase in computational complexity).

All shared work queues are bounded (they serve to achieve load balancing – when full, state handoffs can be skipped).

sample output of a dual-core Spin run

```
$ spin -a petersonMK
$ cc -DNOREDUCE -DDUAL_CORE -o pan pan.c
$ ./pan -z10000 -w27
states stored cpu1 308054 cpu2 106219 ratio: 2.9
states matched cpu1 90618 cpu2 43409 ratio: 2.1
(Spin Version 4.3.0 -- 8 October 2006)
  + Dual Core Processing
  + Partial Order Reduction
hash-Compact 4 search for:
  - never claim             - (none specified)
  - assertion violations    +
  - cycle checks            - (disabled by -DSAFETY)
  - invalid end states      +
State-vector 44 byte, depth reached 10000, errors: 0
548360 transitions + storedmatched
0 atomic steps
hash conflicts: 145 (resolved)
Stats on memory usage (in Megabytes):
23.199 equivalent memory usage for states (storeP(State-vector + overhead))
10.045 actual memory usage for states (compression): 43.30%
12.140 state-vector as stored = 12 byte + 12 byte overhead
1073.742 memory used for hash table (=207)
1024.000 memory used for shared work-queues
1073.742 other (proc and chan stacks)
1033.529 total actual memory usage
unreached in prototype:
line 57, state 30, "and"
1 of 30 states
cpu1: done, 704 Mb of shared state memory left
```
state handoff heuristics for liveness properties

- any “irreversible transition” in the state reachability graph can serve to split the state space
  - separates state space into disjoint parts
  - these transitions can be used to define state handoff points
- trivial application to Spin’s nested depth-first search algorithm for proving liveness:
  - the handoff point is the start of the nested search
  - state spaces can be non-shared (since they are disjoint anyway)
  - should give an immediate (nearly) 2-fold speedup on dual-core systems for all liveness properties

for an irreversible transition there are no return edges across the handoff point: the two parts of the state reachability graph are disjoint

state handoff heuristics for safety properties

- what if there is no suitable irreversible transition?
- we want to achieve:
  - load balancing, but retain the benefits of depth-first search and change as little as possible in the search algorithms in Spin
  - sufficient decoupling of cpu’s (a cpu should be able to do at least N steps with a newly received state, before it hands it off again)
- heuristic used: a handoff depth of modulo N steps (e.g., N: 10...1000)
  - method is intuitively simple
  - giving user control over load-balancing
- generalizes to N-core systems
  - should give near N-fold speedups on N cores

using a shared hash-table each cpu builds a dfs-stack of N steps and then hands off any successor at level N+1
performance of this method

model: leader election in a uni-directional ring (Dolev, Klawe & Rodeh 1982)
problem size: 7 nodes in ring (723K reachable states without p.o. reduction)
comparison of runtime requirements for safety (left) and liveness (right):
  single-core standard Spin verification blue
dual-core verification new algorithm green

---

safety only

* assertions, freedom of deadlock, etc. (with a fixed handoff depth)

liveness

* []<>oneleader (never claim and nested dfs increase runtime)

---

sensitivity to the chosen handoff depth

the characteristic bathtub curve
distributed termination detection


mtype = { Query, Quit, Work };
chan q[2] = [32] of { mtype, byte };
active [2] proctype N() { // the algorithm itself: */
  bool done = false;
  byte s, r, n;
  assert(_pid == 0 || _pid == 1);
  q[1 - _pid]!Work,0; s++; /* seed work items */
  accept: do /* the algorithm itself: */
    :: q[_pid]?Work,0 -> r++; // do forever
    if
      :: (n < 16) -> q[1 - _pid]!Work,0; s++ // only node 0 can initiate termination */
      :: true
      f1
      :: empty(q[0]) && _done && _pid == 0 -> /* node 0 can terminate */
        done = true; /* remember that we sent the Query msg */
        q[1]!Query,s
      :: q[_pid]?Quit,0 -> /* node 1 can now terminate */
        assert(_pid == 1);
        break
      :: q[_pid]?Query,n -> /* only node 1 receives this */
        if
          :: _pid == 1 -> q[0]!Query,r /* respond to termination query from 0 */
          :: _pid == 0 -> /* process response to our termination query */
            if
              :: n == s -> q[1]!Quit,0; break /* accepted; node 0 terminates */
              :: else -> done = false /* try again */
            fi
          fi
        od;
    f1
    assert(empty(q[_pid]))
  od;
}

Peterson's mutual exclusion algorithm (1981)

bool turn, flag[2];
byte ncrit;
active [2] proctype user() /* two processes */
  { // do forever
    :: flag[_pid] = 1; turn = _pid;
    do
      :: flag[1 - _pid] = 0 -> /* wait */
      if
        :: turn != 1 - _pid
        :: else -> break
      fi
      :: else -> break
    od;
    ncrit++;
    assert(ncrit == 1); /* in critical section */
    ncrit--;
    flag[_pid] = 0
  od
}

peterson.c:

#define MB() __asm__ __volatile__ ("mfence":: "memory")
MB();
while (*sh_flag1 == 1 && *sh_turn == 1)
  
  MB();

a fix: add memory barriers
the alternative….

```c
int
tas(volatile int *s)
{
    int r;

    __asm__ __volatile__ (
        "xchgl %0, %1
         : "=r"(r), "=m"(*s)
         : "0"(1), "m"(*s)
         : "memory");

    return r;
}
```

Ugly, but it works, and is fast

Introduces a first platform dependency: different definition of the test&set instruction for each CPU-type (luckily there aren’t many different CPU types in use today)

more examples (dual-core – i.e., the maximal reduction is 50%)
data for runs without partial order reduction – to secure identical state space sizes are explored

fixed handoff depth – safety properties only

---

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Single-Core</th>
<th>Dual-Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peterson’s algorithm N=4</td>
<td>172.85</td>
<td>88.19</td>
</tr>
<tr>
<td>Sliding window protocol W=5</td>
<td>39.73</td>
<td>23.17</td>
</tr>
<tr>
<td>Leader election protocol N=7</td>
<td>174.75</td>
<td>93.00</td>
</tr>
<tr>
<td>Dining philosophers N=9</td>
<td>39.73</td>
<td>23.17</td>
</tr>
<tr>
<td>Peterson’s algorithm N=4</td>
<td>172.85</td>
<td>88.19</td>
</tr>
<tr>
<td>Sliding window protocol W=5</td>
<td>39.73</td>
<td>23.17</td>
</tr>
<tr>
<td>Leader election protocol N=7</td>
<td>174.75</td>
<td>93.00</td>
</tr>
<tr>
<td>Peterson’s algorithm N=4</td>
<td>172.85</td>
<td>88.19</td>
</tr>
<tr>
<td>Sliding window protocol W=5</td>
<td>39.73</td>
<td>23.17</td>
</tr>
<tr>
<td>Leader election protocol N=7</td>
<td>174.75</td>
<td>93.00</td>
</tr>
<tr>
<td>Peterson’s algorithm N=4</td>
<td>172.85</td>
<td>88.19</td>
</tr>
</tbody>
</table>

runtime 42% reduced

runtime 47% reduced

runtime 31% reduced

runtime 36% reduced
hypothesis: the gain for un-optimized code will be larger than for optimized code

adding partial order reduction
the cycle proviso

- to avoid infinite deferral of transitions (the infamous ignoring problem) the standard algorithm checks if any successors are on the dfs stack (the "cycle proviso")
- but we don’t have a full dfs stack in multi-core searches – the stack is split across two or more cpus
- two modifications of the cycle proviso are sufficient to restore soundness and completeness:* )
  1. a full expansion of successor states is done for each ‘border state’ (since we cannot tell if the handed off states are on the stack)
  2. previously visited states that are generated by any cpu with a lower pid, are treated as if they are on the dfs stack
- the cycle proviso works as before elsewhere in the search

*) formal proof courtesy Dragan Bosnacki
dining philosophers
with and without partial order reduction

no major differences
(the partial order reduction algorithm is not very effective on this particular problem)

another example: Peterson’s algorithm
with and without partial order reduction (logscales)

a surprise: partial order reduction can make the advantage of dual-core processing disappear but why?
a reference model

```c
#define BranchSize 8
#define StateSize 500
#define TransTime 9 /*  9 = 1 usec ; 13 = 16 usec */
#define NStates 500000

int count;
byte filler[StateSize];

active [BranchSize] proctype test()
{
end: do
  :: d_step {
    count < NStates ->
    c_code {
      int xi; /* transition delay */
      for (xi = 0; xi < (1<<TransTime); xi++)
      {
        now.filler[xi%StateSize] += xi%256;
      }
      memset(now.filler, 0, StateSize*sizeof(char));
    }
    count++
  }
}
```

study effect of:
branch factor
state size
transition time

measurements dual:single ratios (best value is 0.5)

Reference Model (Transition Delay and BranchFactor Affect Speedup)

Reference Model (Transition Delay and BranchFactor Affect Speedup)

Reference Model (For Small Transition Delays, Statesizes affect Achievable Speedup)

Reference Model (For Larger Transition Delays, Statesizes do not affect Achievable Speedup)
synopsis

- multi-core algorithms do best for verification problems with:
  - larger state sizes (over 100 bytes)
  - larger branch factors (lots of non-determinism)
  - long transition delays (e.g., embedded C-code)

- they give no performance improvement for:
  - small state sizes (less than 100 bytes)
  - small branch factors (less than 2)
  - short transition delays (less than 1 μsec)

- there are cases where a multi-core model checking algorithm cannot compete with a well-tuned single-core model checker
  - e.g., deterministic, models – irrespective of state space size or number of CPU cores...
  - search and compilation optimization can reduce the benefit of multi-core model checking (i.e., they benefit single-core algorithms)
  - specifically, partial order reduction methods reduce the benefit of distributed model checking

- next challenge: is there an efficient (N>2)-core liveness verification algorithm...?