The Design of a Distributed Model Checking Algorithm for

Gerard J. Holzmann

Presented at FMCAD 2006,
San Jose, California
November 14, 2006
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>copying 10 Kb</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM to RAM (memcpy)</td>
<td>3 μsec</td>
</tr>
<tr>
<td>RAM to network port</td>
<td>600 μsec</td>
</tr>
</tbody>
</table>

**relevant factors**

<table>
<thead>
<tr>
<th>Model Checker Performance:</th>
<th>Multi-Core PC (Shared memory)</th>
<th>CPU-cluster (Distributed memory)</th>
</tr>
</thead>
<tbody>
<tr>
<td>relative time to transfer a state to another CPU</td>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>relative time to generate a new state &amp; check if it is previously visited</td>
<td>Fast mc: 10  Slow mc: 50</td>
<td>Fast mc: 10  Slow mc: 50</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)
sample output of a dual-core Spin run

```
$ spin -a petersonN
$ 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
414273 states, stored
134027 states, matched
548300 transitions (= stored+matched)
0 atomic steps
hash conflicts: 145 (resolved)

Stats on memory usage (in Megabytes):
23.199 equivalent memory usage for states (stored*(State-vector + overhead))
10.045 actual memory usage for states (compression: 43.30%)
State-vector as stored = 12 byte + 12 byte overhead
1073.742 memory used for hash table (-w27)
1296.000 memory used for DFS stack (-m27000000)
1024.000 memory used for shared work-queues
1073.741 other (proc and chan stacks)
3453.529 total actual memory usage

unreached in proctype user
  line 57, state 30, "-end-
  1 of 30 states
cpu1: done, 706 Mb of shared state memory left
```

FMCAD2006
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

![Safety Properties -- without partial order reduction](image1)

Liveness

![Liveness Property -- without partial order reduction](image2)

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

[]<>oneleader
(never claim and nested dfs increase runtime)
sensitivity to the chosen handoff depth
the characteristic bathtub curve

Peterson's Algorithm N=4
MaxDepth SearchTree 2,770,018
SV 48

Handoff depth

runtime (seconds)

100%
best

single-core
dual core
distributed termination detection


mtype = { Query, Quit, Work };

chan q[2] = [32] of { mtype, byte };

active [2] proctype N()
{
    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++;
        if
            :: (n < 16) -> q[1 - _pid]!Work,0; s++
            :: true
        fi
        :: empty(q[0]) && !done && _pid == 0 ->
            done = true;
            q[1]!Query,s
        :: q[_pid]?Quit,0 ->
            assert(_pid == 1);
            break
            /* node 1 can now terminate */
        :: q[_pid]?Query,n ->
            if
                :: _pid == 1 -> q[0]!Query,r
                :: _pid == 0 ->
                    if
                        :: n == s -> q[1]!Quit,0; break
                        :: else -> done = false
                    fi
                    /* accept; node 0 terminates */
            fi
            /* respond to termination query from 0 */
            /* process response to our termination query */
        od;
    assert(empty(q[_pid]))
}
Peterson’s mutual exclusion algorithm  (1981)

```c
bool turn, flag[2];
byte ncrit;

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

Surprise: a straight C implementation does not necessarily guarantee mutual exclusion.

A reference implementation in C on a 3.2 GHz dual-core Intel Pentium D – reveals a low probability of mutex violations… (~ 1 in 10^6).

It is caused by out of order execution optimization in the chip itself (not visible in the assembly code).

```
peterson.c: a fix: add memory barriers

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

Many modern CPUs reorder instructions executed and memory accesses to improve execution efficiency. Such processors inherently give some way to force ordering in a stream of memory accesses, typically through a memory barrier instruction. Implementation of Peterson’s and related algorithms on an out-of-order processor generally requires use of such operations to work correctly, but operations from happening in an incorrect order.

Most such CPUs also have a set of guaranteed atomic operations, such as XCHG on x86 processors and Load-Store-Conditional on Alpha, MIPS, PowerPC, and other architectures. These instructions are intended to provide a way to build synchronization primitives more efficiently than can be done with pure software approaches.

See also
- Dekker’s algorithm
- Lamport’s binary algorithm
the alternative....

```c
int
	tas(volatile int *s)
{
    int r;
    __asm__ 
      volatile (__asm__
        "xchgl %0, %1 \n\t"
        : "=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
hypothesis: the gain for un-optimized code will be larger than for optimized code

42% reduction – standard compilation

34% reduction – optimized –O2
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

with partial order reduction

without

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)

without partial order reduction

with partial order reduction

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++
        }
    od
}

study effect of:
branch factor
state size
transition time
```
measurements dual:single ratios (best value is 0.5)
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...?