Multi-core and/or Symbolic Model Checking

Jaco van de Pol
September 20, 2012

Joint work with Stefan Blom, Tom van Dijk, Alfons Laarman, Elwin Pater, and Michael Weber
Model Checking

- Investigate properties of all paths in some discrete graph
- Typically, the graph describes complex system behaviour

Applications
Model Checking

- Investigate properties of all paths in some discrete graph
- Typically, the graph describes complex system behaviour

Applications

- **Safety-critical systems**: prove absence of errors
- **Software validation**: find bugs, generate test cases
  - Indispensable for distributed and multi-core applications
- **Hardware verification**: partly replacing testing @ Intel
Model Checking

- Investigate properties of all paths in some discrete graph
- Typically, the graph describes complex system behaviour

Applications

- **Safety-critical systems**: prove absence of errors
- **Software validation**: find bugs, generate test cases
  - Indispensable for distributed and multi-core applications
- **Hardware verification**: partly replacing testing @ Intel
- **Scheduling/planning**: find the best possible schedule
- **Performance and dependability evaluation**:
  - scenario exhibiting the worst-case latency
  - average latency, long run throughput, mean-time to failure
- **Biology**: investigate dynamic system behaviour *in silico*
State Space Explosion

A combinatorial problem

- number of components
- number of data values
- length of buffers

What if the graph $> 10^{10}$ states?
State Space Explosion

A combinatorial problem

- number of components
- number of data values
- length of buffers

What if the graph > $10^{10}$ states?

- Start computing on an initial part. on-the-fly model checking
- Devise a concise representation. symbolic model checking
- Explore only a subset of transitions. partial order reduction
- Exhibit structural symmetries. symmetry reduction
- Use a cluster of computers. distributed model checking
- Shared-memory solution. multi-core model checking
Requirements for an interface

Access to high-performance model checking

- Technique X is only available for specification language Y
- A user needs to rewrite his model into different languages

![Diagram showing relationships between mCRL2, Promela, UPPAAL, and their reachability and core reachability features.]

- mCRL2
  - Process algebra
- Promela
  - (Spinja)
- UPPAAL
  - (DBM lib)

- PINS
  - Distributed Reachability
  - Multi-core Reachability
  - Symbolic Reachability
Requirements for an interface

Access to high-performance model checking

- Technique X is only available for specification language Y
- A user needs to rewrite his model into different languages

Locality of transitions

- Locality is a key for all high-performance algorithms
- Interface should be simple and generic: black-box
- Interface should expose structural locality: white-box
What is PINS?

**Partitioned Interface for Next States:**
- States are partitioned into vector of \( N \) state parts
- Partition the Next-State function into \( M \) transition groups
- Provide a priori a (hopefully sparse) \( N \times M \) dependency matrix (indicates which state parts each transition group depends on)

On-the-fly access to the state space and locality information
What is PINS?

Partitioned Interface for Next States:

- States are partitioned into vector of $N$ state parts
- Partition the Next-State function into $M$ transition groups
- Provide a priori a (hopefully sparse) $N \times M$ dependency matrix (indicates which state parts each transition group depends on)

On-the-fly access to the state space and locality information

Three basic functions

- GET-MATRIX: returns the dependency matrix $D_{M \times N}$
- INIT-STATE(): returns the initial state vector
- NEXT-STATE(I,S): successors of state $s$ in transition group $i$
LTSmin tool Architecture and Functionality

Analysis algorithms

- Check errors/goals (deadlocks, invariants, actions) on-the-fly
- Distributed generation and minimization modulo bisimulation
- Multi-core LTL model checking
- Symbolic CTL* and μ-calculus model checking
- Disclaimer: partial-order reduction needs a more refined API
# Table of Contents

1. Introduction
   - Model Checking
   - LTSmin toolset

2. Symbolic Model Checking
   - Local Transition Caching
   - Multi-valued Decision Diagrams

3. Multi-core Model Checking
   - Lockless Shared Hashtable
   - State Space Compression
   - Parallel Nested Depth First Search for LTL model checking

4. Symbolic Model Checking with Multi-core BDDs
   - Multi-core Data Structures
   - Parallel BDD operations

5. Conclusion
Example Dependency Matrix

```
int x=7;
process p1() {
  do
    ::{x>0 => x--;y++}
    ::{x>0 => x--;z++}
  od
}
```

```
<table>
<thead>
<tr>
<th></th>
<th>p1</th>
<th>p2</th>
<th>p3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>y</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>z</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
</tbody>
</table>
```
### Example Dependency Matrix

```cpp
int x=7;
process p1() {
  do
    ::{x>0 => x--;y++}
    ::{x>0 => x--;z++}
  od }

int y=3;
process p2() {
  do
    ::{y>0 => y--;x++}
    ::{y>0 => y--;z++}
  od }

int z=9;
process p3() {
  do
    ::{z>0 => z--;x++}
    ::{z>0 => z--;y++}
  od }
```
Example Dependency Matrix

```plaintext
int x=7;
process p1() {
do
::{x>0 => x--;y++}
::{x>0 => x--;z++}
od }

int y=3;
process p2() {
do
::{y>0 => y--;x++}
::{y>0 => y--;z++}
od }

int z=9;
process p3() {
do
::{z>0 => z--;x++}
::{z>0 => z--;y++}
od }
```

Default Matrix

\[
\begin{array}{ccc}
x & y & z \\
p1 & + & + & + \\
p2 & + & + & + \\
p3 & + & + & + \\
\end{array}
\]
Example Dependency Matrix

```
int x=7;
process p1() {
  do
  ::{x>0 => x--;y++}
  ::{x>0 => x--;z++}
  od }

int y=3;
process p2() {
  do
  ::{y>0 => y--;x++}
  ::{y>0 => y--;z++}
  od }

int z=9;
process p3() {
  do
  ::{z>0 => z--;x++}
  ::{z>0 => z--;y++}
  od }
```

Default Matrix

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p2</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p3</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Better Matrix

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1.1</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>p1.2</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>p2.1</td>
<td>+</td>
<td>+</td>
<td>−</td>
</tr>
<tr>
<td>p2.2</td>
<td>−</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p3.1</td>
<td>+</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>p3.2</td>
<td>−</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Example Dependency Matrix

```
int x=7;
process p1() {
  do ::{x>0 => x--;y++}
      ::{x>0 => x--;z++}
  od }

int y=3;
process p2() {
  do ::{y>0 => y--;x++}
      ::{y>0 => y--;z++}
  od }

int z=9;
process p3() {
  do ::{z>0 => z--;x++}
      ::{z>0 => z--;y++}
  od }
```

Default Matrix

```
\begin{bmatrix}
  x & y & z \\
  p1 & + & + & + \\
  p2 & + & + & + \\
  p3 & + & + & + \\
\end{bmatrix}
```

Better Matrix

```
\begin{bmatrix}
  x & y & z \\
  p1.1 & + & + & - \\
  p1.2 & + & - & + \\
  p2.1 & + & + & - \\
  p2.2 & - & + & + \\
  p3.1 & + & - & + \\
  p3.2 & - & + & + \\
\end{bmatrix}
```

init state = ⟨7, 3, 9⟩

```
⟨7, 3, 9⟩ \xrightarrow{p1.1} ⟨6, 4, 9⟩
⟨7, 3, *⟩ \xrightarrow{p1.1} ⟨6, 4, *⟩
⟨7, 3, 9⟩ \xrightarrow{p3.2} ⟨7, 4, 8⟩
⟨*, 3, 9⟩ \xrightarrow{p3.2} ⟨*, 4, 8⟩
```
**Example Dependency Matrix**

```
int x=7;
process p1() {
do::{x>0 => x--;y++}
::{x>0 => x--;z++}
od }

int y=3;
process p2() {
do::{y>0 => y--;x++}
::{y>0 => y--;z++}
od }

int z=9;
process p3() {
do::{z>0 => z--;x++}
::{z>0 => z--;y++}
od }
```

### Default Matrix

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p2</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p3</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

### Better Matrix

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1.1</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>p1.2</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>p2.1</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>p2.2</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p3.1</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>p3.2</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

### Static Regrouping

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1.1, 2.1</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>p1.2, 3.1</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>p2.2, 3.2</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Caching Local Transitions

- Consider local transition in specification:
  \[ p3.2: \text{atomic} \{ z>0 \rightarrow z--; y++ \} \]

- Dependency matrix row:
  \[
  \begin{array}{ccc}
  x & y & z \\
  \hline
  p3.2 & 0 & 1 & 1 \\
  \end{array}
  \]

- Define projection: \( \pi_{p3.2} \langle x, y, z \rangle = \langle y, z \rangle \)
Caching Local Transitions

- Consider local transition in specification:
  \[ p3.2: \text{atomic} \{ \; z > 0 \rightarrow z--; \; y++ \; \} \]

- Dependency matrix row:
  \[
  \begin{bmatrix}
  x & y & z \\
  \end{bmatrix}
  \]

- Define projection: \( \pi_{p3.2} \langle x, y, z \rangle = \langle y, z \rangle \)

Next, consider two consecutive calls to \( p3.2 \)

- First call: \( \langle x, y, z \rangle \)
- Successor: \( \langle x, y', z' \rangle \)
- Project and store in cache: \( \langle y, z \rangle \rightarrow \langle y', z' \rangle \)
Caching Local Transitions

- Consider local transition in specification:
  \[ p3.2: \text{atomic} \{ z>0 \rightarrow z--; y++ \} \]

- Dependency matrix row:
  \[ p3.2 \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \]

- Define projection: \( \pi_{p3.2} \langle x, y, z \rangle = \langle y, z \rangle \)

Next, consider two consecutive calls to \( p3.2 \)

- first call: \( \langle x, y, z \rangle \)
- successor: \( \langle x, y', z' \rangle \)
- project and store in cache: \( \langle y, z \rangle \rightarrow \langle y', z' \rangle \)
- second call: \( \langle x'', y, z \rangle \)
- project: \( \langle y, z \rangle \)
- cache lookup: \( \rightarrow \langle y', z' \rangle \)
- expand: \( \langle x'', y', z' \rangle \)
Consider local transition in specification:
\[ p3.2: \text{atomic } \{ z > 0 \rightarrow z--; y++ \} \]

Dependency matrix row:
\[ p3.2 \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \]

Define projection:
\[ \pi_{p3.2}(x, y, z) = (y, z) \]

Next, consider two consecutive calls to \( p3.2 \)

- First call: \( (x, y, z) \)
- Successor: \( (x, y', z') \)
- Project and store in cache: \( (y', z') \)
- Second call: \( (x'', y, z) \)
- Project: \( (y, z) \)
- Cache lookup: \( \rightarrow (y', z') \)
- Expand: \( (x'', y', z') \)

Transition caching saves calls to the language module

Memoization table \( cache[i] \) for each transition group \( i \)
Every path in the MDD represents a concrete state vector.
Every path in the MDD represents a concrete state vector

Potential gain in memory saving: exponential (here: $54 \rightarrow 15$)

Symbolic Reachability: explore sets of states stored as MDDs
Symbolic Reachability Algorithm

- $L, V$: MDDs for sets of long state vectors (level, visited)
- $R_i$: MDDs to store transition relation $i$ on short vectors
- $L_i, V_i$: MDDs for sets of short state vectors (level, visited for $i$)
Symbolic Reachability Algorithm

- $L, V$: MDDs for sets of long state vectors (level, visited)
- $R_i$: MDDs to store transition relation $i$ on short vectors
- $L_i, V_i$: MDDs for sets of short state vectors (level, visited for $i$)

**symbolic-reachability: simplest version**

1. $L := \{\text{INITSTATE}()\}; \quad V := L; \quad \text{all } R_i := \emptyset; \quad \text{all } V_i := \emptyset$
2. while $L \neq \emptyset$ do
3.   for $i \in \text{groups}$ do
4.     $L_i := \pi_i([D]_{N \times K}, L) \setminus V_i; \quad V_i := V_i \cup L_i$
5.   $R_i := R_i \cup \{(s, s') | s \in L_i \land s' \in \text{NEXTSTATE}(i, s)\}$
6.   $L := \bigcup_i (\text{RELPROD}(R_i, L) \setminus V); \quad V := V \cup L$
7. return $V$

- **RELPROD** and **OR** are MDD operations
Symbolic Reachability

Symbolic Reachability with PINS

- Global set of reachable states is computed as fix point
- Stored as a multi-valued decision diagram (MDD)
- Learn symbolic sub-groups $R_i$ on-the-fly (via `NEXTSTATE`)

Up to $>2^{50}$ states now also for PROMELA and mCRL2
Symbolic Reachability

Symbolic Reachability with PINS

- Global set of reachable states is computed as fix point
- Stored as a multi-valued decision diagram (MDD)
- Learn symbolic sub-groups $R_i$ on-the-fly (via NEXTSTATE)

Up to $> 2^{50}$ states now also for PROMELA and mCRL2

Extensions

- Multiple exploration strategies:
  - Breadth-first: $(T_1 + T_2 + \cdots + T_n)^*$
  - Chaining: $(T_1 \circ T_2 \circ \cdots \circ T_n)^*$
  - Saturation-like: $(T_1^* \circ T_2^* \circ \cdots \circ T_n^*)^*$
  - Full Saturation: $(((T_1^* T_2^*)^* T_3)^* \cdots)^* T_n)^*$
Symbolic Reachability

Symbolic Reachability with PINS

- Global set of reachable states is computed as fix point
- Stored as a multi-valued decision diagram (MDD)
- Learn symbolic sub-groups $R_i$ on-the-fly (via $\text{NEXT\,STATE}$)

Up to $> 2^{50}$ states now also for PROMELA and mCRL2

Extensions

- Multiple exploration strategies:
  - Breadth-first: $(T_1 + T_2 + \cdots + T_n)^*$
  - Chaining: $(T_1 \circ T_2 \circ \cdots \circ T_n)^*$
  - Saturation-like: $(T_1^* \circ T_2^* \circ \cdots \circ T_n^*)^*$
  - Full Saturation: $(((T_1^* T_2^* T_3^*) \cdots T_n^*)^*)^*$
- Static variable reordering boosts performance
- Multiple BDD packages: BuDDy, own MDD, Sylvan (later)
Multi-core Model Checking
Multi-Core “Crisis”

Recent “standard hardware”

- Multiple processors, multiple cores
  - E.g.: 48 cores, 256GB shared mem
  - L2 caches: cache-coherence
  - Non-Uniform Memory Architecture

(Source: Smoothspan)

Source: Anandtech

Source: Smoothspan
Multi-Core “Crisis”

Recent “standard hardware”

- Multiple processors, multiple cores
- E.g.: 48 cores, 256GB shared mem
- L2 caches: cache-coherence
- Non-Uniform Memory Architecture
# Table of Contents

1. Introduction
   - Model Checking
   - LTSmin toolset

2. Symbolic Model Checking
   - Local Transition Caching
   - Multi-valued Decision Diagrams

3. Multi-core Model Checking
   - Lockless Shared Hashtable
   - State Space Compression
   - Parallel Nested Depth First Search for LTL model checking

4. Symbolic Model Checking with Multi-core BDDs
   - Multi-core Data Structures
   - Parallel BDD operations

5. Conclusion
Multi-core Reachability - Architecture I

- **DiVinE**: Static Partitioning
- **BFS**, communication

- **SPIN**: Shared Storage, Stack Slicing
- **DFS**, integrated load balancing
Multi-core Reachability - Architecture I

- DiVinE: Static Partitioning
- BFS, communication

- SPIN: Shared Storage, Stack Slicing
- DFS, integrated load balancing

Main bottleneck for shared storage: Visited Set

- Parallel access requires synchronization ....... lock contention
- Graph traversal: Random memory access ....... cache misses
Shared state storage as main synchronization point

- Flexible state space traversal \ldots\ \ldots\ \text{free/strict} BFS and DFS
- Flexible load balancing \ldots\ \ldots\ \text{Synchronous Random Polling}
- Single function needed: \textsc{Find-or-Put}
  - Monotonically growing set of visited states
Lockless Hash Table: Layout
[FMCAD 2010]

- Open addressing
- Separate hash and data
- Hash Memoization
- *Walking the Line*
- Lockless (CAS + write bit)

See also: Cliff Click (JavaOne 2007 presentation)
Lockless Hash Table: Layout
[FMCAD 2010]

- Open addressing
- Separate hash and data
- Hash Memoization
- Walking the Line
- Lockless (CAS + write bit)

See also: Cliff Click (JavaOne 2007 presentation)
Further compression: (Parallel) Compact Hash Table
[J.G. Cleary’84, Vegt/Laarman’12]
Experiments from 2010 (BEEM database)

**SPIN 5.2.4 (NASA/JPL)**

**DiVinE 2.2 (Brno,CZ)**

**LTSmin, no load balancing**

**LTSmin, random synchronous polling**
Tree Compression – addressing memory consumption [PDMC 2007, JLC 2009]

- Index 5 to folded vector \( \langle 2, 1 \rangle \) represents state \( \langle 3, 5, 5, 4, 1, 3 \rangle \)
- Selected tree fringe of grey boxes corresponds to state vector
Tree Compression – addressing memory consumption
[PDMC 2007, JLC 2009]

- Index 5 to folded vector \(\langle 2, 1 \rangle\) represents state \(\langle 3, 5, 5, 4, 1, 3 \rangle\)
- Selected tree fringe of grey boxes corresponds to state vector
- Potential gain (here 54 → 42 entries):
  - main table of size \(N\) is only two integers wide
  - small tables have expected size \(O(\sqrt{N})\) only
Exploiting locality once more

Dependency Matrix $D_{M \times N}$ predicts changing state parts:

- Incremental tree insertions:
  - Traverse only the changing paths in the Tree of Tables

Exploiting locality once more

Dependency Matrix $D_{M \times N}$ predicts changing state parts:

- Incremental tree insertions:
  - Traverse only the changing paths in the Tree of Tables
- Incremental hashing, based on Zobrist:

\[
\begin{align*}
H_x & (H_x \oplus Z_{\triangleleft, g, 1}) \oplus Z_{\triangleleft, f, 3} = H_y \\
\text{(Diagram of chessboard changes from g1-f3)}
\end{align*}
\]

Multi-Core Tree Compression – experiments
[NASA FM 2011]

- Tree compression is a recursive variant of SPIN’s COLLAPSE
- Exploit combinatorial structure:
  - State vectors highly similar
  - Impressive compression ratios
- Extreme case: firewire_tree
  - Uncompressed: 14 GB
  - Tree Compression: 96 MB
- Compression comes for free
  - Arithmetic intensity increases
  - Less memory-bus traffic
Model Checking by Accepting Cycles

LTL Model Checking

- A buggy run in a system can be viewed as an infinite word
- Absence of bugs: emptiness of some Büchi automaton
- Graph problem: find a reachable accepting state on a cycle
- Basic algorithm: Nested Depth First Search (NDFS)
LTL Model Checking

- A buggy run in a system can be viewed as an infinite word
- Absence of bugs: emptiness of some Büchi automaton
- Graph problem: find a reachable accepting state on a cycle
- Basic algorithm: Nested Depth First Search (NDFS)
LTL Model Checking

- A buggy run in a system can be viewed as an infinite word
- Absence of bugs: emptiness of some Büchi automaton
- Graph problem: find a reachable accepting state on a cycle
- Basic algorithm: Nested Depth First Search (NDFS)
Model Checking by Accepting Cycles

LTL Model Checking

- A buggy run in a system can be viewed as an infinite word
- Absence of bugs: emptiness of some Büchi automaton
- Graph problem: find a reachable accepting state on a cycle
- Basic algorithm: Nested Depth First Search (NDFS)

New in LTSmin (ATVA’11,’12)

- Scalable parallel NDFS
- So far, thought to be impossible (in theory)
Nested Depth First Search
[Courcoubetis, Vardi, et al.]

procedure $\text{DFSblue}(s)$
    $s.\text{blue} := \text{true}$
    for all $t \in \text{NextState}(s)$ do
        if $\neg t.\text{blue}$ then $\text{DFSblue}(t)$
    if $s \in \text{Accepting}$ then
        seed := $s$
        $\text{DFSred}(s)$

Nested DFS
- Blue search
  - Visits all reachable states
  - Starts Red search on accepting states (seed) in post order
Nested Depth First Search
[Courcoubetis, Vardi, etal.]

**procedure** DFS\textsc{blue}(s)
\begin{align*}
s.\text{blue} & := \text{true} \\
\text{for all } t & \in \text{NextState}(s) \text{ do} \\
& \quad \text{if } \neg t.\text{blue} \text{ then } \text{DFS}\textsc{blue}(t) \\
& \quad \text{if } s \in \text{Accepting} \text{ then} \\
& \qquad \text{seed} := s \\
& \qquad \text{DFS}\textsc{red}(s)
\end{align*}

**procedure** DFS\textsc{red}(s)
\begin{align*}
s.\text{red} & := \text{true} \\
\text{for all } t & \in \text{NextState}(s) \text{ do} \\
& \quad \text{if } t = \text{seed} \text{ then } \text{ExitCycle} \\
& \quad \text{if } \neg t.\text{red} \text{ then } \text{DFS}\textsc{red}(t)
\end{align*}

**Nested DFS**
- **Blue search**
  - Visits all reachable states
  - Starts Red search on accepting states (seed) in post order
- **Red Search**
  - Finds cycle through seed
  - Visits states at most once
Nested Depth First Search
[Courcoubetis, Vardi, etal.]

procedure $\text{DFS\blue}(s)$
    s.blue := true
    for all $t \in \text{NextState}(s)$ do
        if $\neg t$.blue then $\text{DFS\blue}(t)$
        if $s \in \text{Accepting}$ then
            seed := s
            $\text{DFS\red}(s)$

procedure $\text{DFS\red}(s)$
    s.red := true
    for all $t \in \text{NextState}(s)$ do
        if $t = \text{seed}$ then $\text{ExitCycle}$
        if $\neg t$.red then $\text{DFS\red}(t)$

Nested DFS

- Blue search
  - Visits all reachable states
  - Starts Red search on accepting states (seed) in post order

- Red Search
  - Finds cycle through seed
  - Visits states at most once

- Linear time, on-the-fly

- Blue is inherently depth-first
Swarmed Multi-core Nested Depth First Search

code for worker $i$, no communication

**procedure** $\text{DFSBlue}(s,i)$

\begin{align*}
& s.\text{blue}[i] := \text{true} \\
& \text{for all } t \in \text{NextState}(s) \text{ do} \\
& \quad \text{if } \neg t.\text{blue}[i] \text{ then } \text{DFSBlue}(t,i) \\
& \text{if } s \in \text{Accepting} \text{ then} \\
& \quad \text{seed}[i] := s \\
& \quad \text{DFSRed}(s,i)
\end{align*}

**procedure** $\text{DFSRed}(s,i)$

\begin{align*}
& s.\text{red}[i] := \text{true} \\
& \text{for all } t \in \text{NextState}(s) \text{ do} \\
& \quad \text{if } t = \text{seed}[i] \text{ then } \text{ExitCycle} \\
& \quad \text{if } \neg t.\text{red}[i] \text{ then } \text{DFSRed}(t,i)
\end{align*}

Multi-core Swarmed NDFS

- $N$ workers perform parallel search **independently**
  
  [G. Holzmann et al.]
code for worker \( i \), no communication

**procedure** \( \text{DFSblue}(s, i) \)

\[
\text{s.blue}[i] := \text{true}
\]

\[\text{for all } t \in \text{NextState}(s) \text{ do} \]

\[
\text{if } \neg t.\text{blue}[i] \text{ then } \text{DFSblue}(t, i)
\]

\[\text{if } s \in \text{Accepting} \text{ then} \]

\[
\text{seed}[i] := s
\]

\[\text{DFSred}(s, i)\]

**procedure** \( \text{DFSred}(s, i) \)

\[
\text{s.red}[i] := \text{true}
\]

\[\text{for all } t \in \text{NextState}(s) \text{ do} \]

\[
\text{if } t = \text{seed}[i] \text{ then } \text{ExitCycle}
\]

\[
\text{if } \neg t.\text{red}[i] \text{ then } \text{DFSred}(t, i)
\]

**Multi-core Swarmed NDFS**

- \( N \) workers perform parallel search independently
  
  \[\text{[G. Holzmann etal.]}\]

- Multi-core: store visited states in a shared hash table
Swarmed Multi-core Nested Depth First Search

code for worker \( i \), no communication

\begin{align*}
\textbf{procedure} & \quad \text{DFSblue}(s, i) \\
& \quad s.\text{blue}[i] := \text{true} \\
& \quad \text{for all } t \in \text{NextState}(s) \text{ do} \\
& \quad \quad \text{if } \neg t.\text{blue}[i] \text{ then } \text{DFSblue}(t, i) \\
& \quad \text{if } s \in \text{Accepting} \text{ then} \\
& \quad \quad \text{seed}[i] := s \\
& \quad \quad \text{DFSred}(s, i)
\end{align*}

\begin{align*}
\textbf{procedure} & \quad \text{DFSred}(s, i) \\
& \quad s.\text{red}[i] := \text{true} \\
& \quad \text{for all } t \in \text{NextState}(s) \text{ do} \\
& \quad \quad \text{if } t = \text{seed}[i] \text{ then } \text{ExitCycle} \\
& \quad \quad \text{if } \neg t.\text{red}[i] \text{ then } \text{DFSred}(t, i)
\end{align*}

Multi-core Swarmed NDFS

\begin{itemize}
\item \( N \) workers perform parallel search \textit{independently} \[\text{G. Holzmann etal.}\]
\item \textbf{Multi-core}: store visited states in a shared hash table
\item Scales well in the presence of accepting cycles (bugs)
\item Otherwise, all workers traverse the whole graph
\end{itemize}
Approaches to Parallel LTL Model Checking

Speedup of Swarmed NDFS (1 versus 16 cores)

Alternatives for parallel LTL

- Swarm verification with NDFS
  - Effective, only for bug finding

[BEEM database]
Approaches to Parallel LTL Model Checking

Speedup of Swarmed NDFS
(1 versus 16 cores)

Alternatives for parallel LTL

- **Swarm verification** with NDFS
  - Effective, only for bug finding
- **Dual-core NDFS** [Holzmann]
  - Red search on 2nd CPU
  - Speedup of at most factor 2

[BEEM database]
Approaches to Parallel LTL Model Checking

Speedup of Swarmed NDFS
(1 versus 16 cores)

Alternatives for parallel LTL

- Swarm verification with NDFS
  - Effective, only for bug finding
- Dual-core NDFS [Holzmann]
  - Red search on 2nd CPU
  - Speedup of at most factor 2
- Red Search as parallel reachability
  - Speedup still $\leq 2:\ |G| + |G|/N$

[BEEM database]
Approaches to Parallel LTL Model Checking

Speedup of Swarmed NDFS  
(1 versus 16 cores)

Alternatives for parallel LTL

- **Swarm verification** with NDFS
  - Effective, only for bug finding
- **Dual-core NDFS**  
  - Red search on 2nd CPU
  - Speedup of at most factor 2
- **Red Search as parallel reachability**
  - Speedup still $\leq 2: |G| + |G|/N$
- **Can one do better?**
  - Post-order is P-Complete, so
  - DFS not efficiently parallelizable

[BEEM database]
**Approaches to Parallel LTL Model Checking**

**Speedup of Swarmed NDFS**
(1 versus 16 cores)

![Graph showing speedup of swarmed NDFS](image)

**Alternatives for parallel LTL**

- **Swarm verification** with NDFS
  - Effective, only for bug finding
- **Dual-core NDFS** [Holzmann]
  - Red search on 2nd CPU
  - Speedup of at most factor 2
- **Red Search as parallel reachability**
  - Speedup still ≤ 2: $|G| + |G|/N$
- **Can one do better?**
  - Post-order is P-Complete, so
  - DFS not efficiently parallelizable
- **Breadth-first based:**
  - **OWCTY, MAP** [Brno]
  - Not linear ($|G| \cdot h$), not on-the-fly

[BEEM database]
Parallel NDFS: share the red color
[ATVA’11]

procedure DFS\textsc{blue}(s,i)
\begin{align*}
&\text{s.color}[i] := \text{cyan} \\
&\text{for all } t \in \text{NextState}(s) \text{ do} \\
&\quad \text{if } t.\text{color}[i] = \text{cyan} \text{ and } s \text{ or } t \in \text{Acc} \text{ then } \text{ExitCycle} \\
&\quad \text{if } t.\text{color}[i] = \text{white} \text{ and } \neg t.\text{red} \text{ then } \text{DFS\textsc{blue}}(t,i) \\
&\quad \text{if } s \in \text{Acc} \text{ then } s.\text{count}++; \text{ DFS\textsc{red}}(s,i) \\
&\text{s.color}[i] := \text{blue}
\end{align*}

procedure DFS\textsc{red}(s,i)
\begin{align*}
&\text{s.color}[i] := \text{pink} \\
&\text{for all } t \in \text{NextState}(s) \text{ do} \\
&\quad \text{if } t.\text{color}[i] = \text{cyan} \text{ then } \text{ExitCycle} \\
&\quad \text{if } t.\text{color}[i] \neq \text{pink} \text{ and } \neg t.\text{red} \text{ then } \text{DFS\textsc{red}}(t,i) \\
&\quad \text{if } s \in \text{Acc} \text{ then } s.\text{count}--; \text{ await } s.\text{count}=0 \\
&\text{s.red} := \text{true}
\end{align*}
Parallel NDFS: share the red color
[ATVA'11]

**procedure DFSBLUE**(s,i)

s.color[i] := cyan

for all t ∈ **NEXTSTATE**(s) do

    if t.color[i] = cyan and s or t ∈ Acc then ExitCycle

    if t.color[i] = white and ¬t.red then **DFSBLUE**(t,i)

if s ∈ Acc then s.count++; **DFSRED**(s,i)

s.color[i] := blue

**procedure DFSRED**(s,i)

s.color[i] := pink

for all t ∈ **NEXTSTATE**(s) do

    if t.color[i] = cyan then ExitCycle

    if t.color[i] ≠ pink and ¬t.red then **DFSRED**(t,i)

if s ∈ Acc then s.count--; **await** s.count=0

s.red := true
**OWCTY and Swarmed NDFS versus Parallel NDFS**

**Swarmed versus Parallel NDFS**
(both 16 cores)

**OWCTY versus Parallel NDFS**
(both 16 cores)
Multi-core Symbolic Model Checking
Observations

- Time-wise, multicore model checking yields great speedups
- Memory-wise, BDDs are potentially exponentially concise
- Can we design scalable parallel symbolic data structures?
Observations

- Time-wise, multicore model checking yields great speedups
- Memory-wise, BDDs are potentially exponentially concise
- Can we design scalable parallel symbolic data structures?
- Memory-intensive, random memory access, cannot scale
Earlier work: disappointing speedups

Earlier work in parallel BDDs

- early ’90s: vector machines, massive SIMD (not unlike GPU)
- late ’90s: virtual SMP, distributed BDDs (BDDnow)
Earlier work: disappointing speedups

**Earlier work in parallel BDDs**
- early '90s: vector machines, massive SIMD (not unlike GPU)
- late '90s: virtual SMP, distributed BDDs (BDDnow)

**Earlier work in parallel symbolic model checking**
- '00vv: Grumberg et al: vertical splitting of BDDs (distributed)
- '00vv: Ciardo et al: horizontal splitting of BDDs (distributed)
- CAV'07: Lüttgen et al - parallelisation of saturation with Cilk
- PDMC’09: Ciardo - **Difficult, but what is the alternative?**
# Earlier work: disappointing speedups

## Earlier work in parallel BDDs
- early '90s: vector machines, massive SIMD (not unlike GPU)
- late '90s: virtual SMP, distributed BDDs (BDDnow)

## Earlier work in parallel symbolic model checking
- '00vv: Grumberg et al: vertical splitting of BDDs (distributed)
- '00vv: Ciardo et al: horizontal splitting of BDDs (distributed)
- CAV’07: Lüttgen etal - parallelisation of saturation with Cilk
- PDMC’09: Ciardo - **Difficult, but what is the alternative?**

## Recent developments
- 2010: J. Ossowski - Jinc: Multi-threaded decision diagrams
- 2012: Tom van Dijk, Alfons Laarman, JvdP: **Sylvan, a library for multi-core BDD operations (PDMC’12)**
## Table of Contents

1. **Introduction**
   - Model Checking
   - LTSmin toolset

2. **Symbolic Model Checking**
   - Local Transition Caching
   - Multi-valued Decision Diagrams

3. **Multi-core Model Checking**
   - Lockless Shared Hashtable
   - State Space Compression
   - Parallel Nested Depth First Search for LTL model checking

4. **Symbolic Model Checking with Multi-core BDDs**
   - Multi-core Data Structures
   - Parallel BDD operations

5. **Conclusion**
Reuse lockless Hash Tables?

Data structures for Binary Decision Diagrams

- **Unique Table**: ensure that BDD nodes exist at most once
  - implements the *maximal sharing* requirement
  - necessary for constant *equality check* by pointer comparison
- **Computed Table**: dynamic programming
  - Used to store intermediate results to avoid recomputations
  - Needed to manipulate BDDs in polynomial time
Reuse lockless Hash Tables?

Data structures for Binary Decision Diagrams

- **Unique Table:** ensure that BDD nodes exist at most once
  - implements the *maximal sharing* requirement
  - necessary for constant *equality check* by pointer comparison
- **Computed Table:** dynamic programming
  - Used to store intermediate results to avoid recomputations
  - Needed to manipulate BDDs in polynomial time

- Both data structures are usually implemented as *hash tables*
- **Problem:** Both tables require *garbage collection* in practice
Algorithm to put an entry in Computed Table

- Local bit-lock in hash array controls access to data array
- Every cache-hit is nice, but don't loose time
  - No waiting for locks: just give up
  - No hash collision resolution: just overwrite or give up
Algorithm to put an entry in Computed Table

- Local bit-lock in hash array controls access to data array
- Every cache-hit is nice, but don’t loose time
  - No waiting for locks: just give up
  - No hash collision resolution: just overwrite or give up

**Input:** key, data

1: \(\text{hash} \leftarrow \text{calculate}\_\text{hash}(\text{key})\)
2: \(\text{index} \leftarrow \text{hash} \% \text{tablesize}\)
3: \(\langle \text{curhash}, \text{curlock} \rangle \leftarrow \text{hasharray}[\text{index}]\)
4: \(\textbf{if curlock = 1 then return NOTADDED}\)
5: \(\textbf{if curhash = hash then}\)
6: \(\textbf{if data matches data in data array then return NOTADDED}\)
7: \(\textbf{if not compare\_and\_swap(hasharray[index], \langle \text{curhash}, 0 \rangle, \langle \text{hash}, 1 \rangle) then}\)
8: \(\textbf{return NOTADDED}\)
9: \(\text{write data to data array}\)
10: \(\text{hasharray}[\text{index}] \leftarrow \langle \text{hash}, 0 \rangle \{\text{release lock}\}\)
11: \(\textbf{return ADDED}\)
Unique table = lockless hashtable + garbage collection

- Use one bit of hash as lock, and two bytes as reference count
Unique table = lockless hashtable + garbage collection

- Use one bit of hash as lock, and two bytes as reference count
- Buckets in the hash table are in one of these states:
  1. **EMPTY**: Unused bucket, also end-of-search
  2. **TOMBSTONE**: Unused bucket, but keep searching
  3. **WAIT(hash)**: Bucket being written
  4. **DONE(hash, count)**: Filled bucket

![Diagram](attachment:unique_table_diagram.png)
Unique table = lockless hashtable + garbage collection

- Use one bit of hash as lock, and two bytes as reference count
- Buckets in the hash table are in one of these states:
  1. EMPTY: Unused bucket, also end-of-search
  2. TOMBSTONE: Unused bucket, but keep searching
  3. WAIT(hash): Bucket being written
  4. DONE(hash, count): Filled bucket

Rules to enforce correctness:
1. Inserting and deleting mutually exclusive (separate gc mode)
2. Transitions to WAIT use compare_and_swap
Parallel BDD operations

- **RelProd** and **OR** have two recursive calls
- Calls organized in a *task dependency graph*
Parallel BDD operations

- **RelProd** and **Or** have two recursive calls
- Calls organized in a *task dependency graph*
- Fine-grained task-parallelism:
  - Parent spawns children for subtasks
  - Parent waits upon completion of children
  - Load balancing by work-stealing
  - **Cilk** [Blumofe ’95] or **Wool** [Faxén 2008]
- **Alternative**: rely on random traversal order
Parallel BDD operations

- **RELPROD** and **OR** have two recursive calls
- Calls organized in a *task dependency graph*
- Fine-grained task-parallelism:
  - Parent spawns children for subtasks
  - Parent waits upon completion of children
  - Load balancing by work-stealing
  - **Cilk** [Blumofe ’95] or **Wool** [Faxén 2008]
- **Alternative**: rely on random traversal order
- Same task might be created **many times**
- Store the results in a memoization table
Results: speedup of BDD operations for model checking

Conclusion

- BEEM benchmarks, again
- On $4 \times 12 = 48$ core NUMA
- Speedup of up to 32
- Small models don’t scale (time spent in work stealing)
Results: speedup of BDD operations for model checking

![Graph showing speedup of BDD operations across different models and workers.]

### Conclusion

- **BEEM benchmarks, again**
- **On $4 \times 12 = 48$ core NUMA**
- **Speedup of up to 32**
- **Small models don’t scale**
  (time spent in work stealing)

- We only speedup the **BDD-operations** in symbolic reachability
- Measurements exclude time spent in **language modules**
- Even for large models, many small BDDs are involved
- **This has potential impact** on hardware verification
Table of Contents

1. Introduction
   - Model Checking
   - LTSmin toolset

2. Symbolic Model Checking
   - Local Transition Caching
   - Multi-valued Decision Diagrams

3. Multi-core Model Checking
   - Lockless Shared Hashtable
   - State Space Compression
   - Parallel Nested Depth First Search for LTL model checking

4. Symbolic Model Checking with Multi-core BDDs
   - Multi-core Data Structures
   - Parallel BDD operations

5. Conclusion
Lessons learnt

Data structures: engineering
- Lockless, Compare-and-Swap
- Tedious > 1 year: know your hardware
- Correctness? We used our tools:
  - mCRL2/LTSmin: algorithms
  - Spin/LTSmin: C-code level
  - Separation logic (ongoing)
Lessons learnt

Data structures: engineering
- Lockless, Compare-and-Swap
- Tedious > 1 year: know your hardware
- Correctness? We used our tools:
  - mCRL2/LTSmin: algorithms
  - Spin/LTSmin: C-code level
  - Separation logic (ongoing)

Algorithms: ingenuity
- Very simple architecture
- Workers mostly independent
  - Rely on randomness
  - Some global info shared
- Rigorous mathematical proof
  - (only partly in PVS)
Lessons learnt

Data structures: engineering
- Lockless, Compare-and-Swap
- Tedious > 1 year: know your hardware
- Correctness? We used our tools:
  - mCRL2/LTSmin: algorithms
  - Spin/LTSmin: C-code level
  - Separation logic (ongoing)

Algorithms: ingenuity
- Very simple architecture
- Workers mostly independent
  - Rely on randomness
  - Some global info shared
- Rigorous mathematical proof
  - (only partly in PVS)

Performance: repeatable experiments, open data
- Measure, measure, measure, hypothesize, visualize
- BEEM: > 250 Benchmarks for Explicit MC; Spin models
- Possible bias to our 16-core and 48-core AMD machines
- Applications: Railway Interlockings, CERN LHC control
- Todo: hardware benchmarks for symbolic model checking
Check out: http://fmt.cs.utwente.nl/tools/ltsmin/

- Tom van Dijk, Alfons Laarman and Jaco van de Pol, *Multi-core BDD Operations for Symbolic Reachability* .... (PDMC 2012)
- Alfons Laarman, Rom Langerak, Jaco vd Pol, Michael Weber, A. Wijs, *Multi-Core Nested Depth-First Search* .... (ATVA 2011,’12)
- Alfons Laarman, Jaco van de Pol, Michael Weber, *Parallel Recursive State Compression for Free* .... (SPIN 2011)
- Alfons Laarman, Jaco van de Pol, Michael Weber, *Multi-Core LTSmin: Marrying Modularity and Scalability* .... (NFM 2011)
- Stefan Blom, Jaco van de Pol, Michael Weber, *LTSmin: Distributed and Symbolic Reachability* .... (CAV 2010)
- Alfons Laarman, Jaco van de Pol and Michael Weber, *Boosting Multi-Core Reachability Performance with Shared Hash Tables*
- Stefan Blom, Bert Lisser, Jaco van de Pol and Michael Weber, *A Database Approach to Distributed State-Space Generation* (JLC 2009)