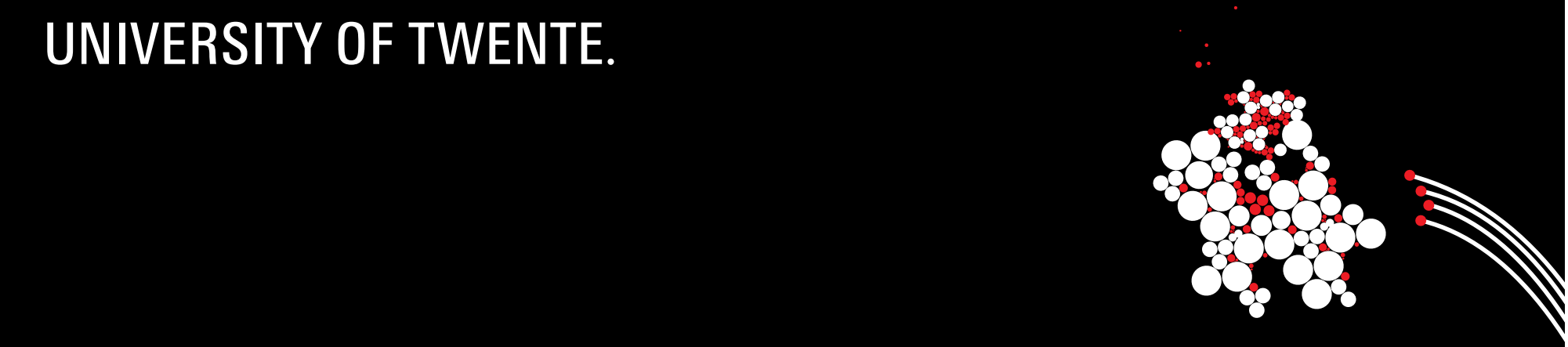
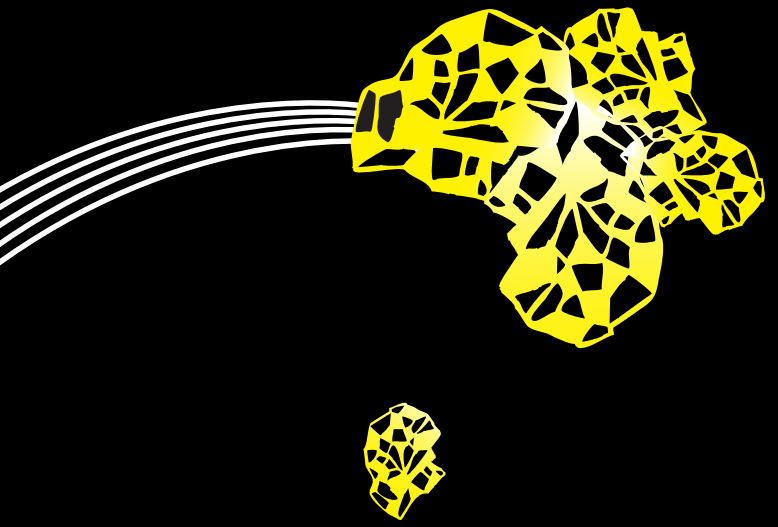
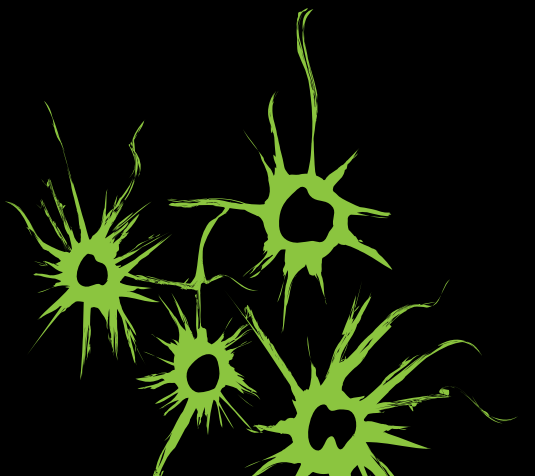
UNIVERSITY OF TWENTE.



Sylvan: Multi-core Decision Diagrams

Tom van Dijk & Jaco van de Pol
Formal Methods and Tools
University of Twente, The Netherlands





Sylvan: Multi-core Decision Diagrams

This is about...

- Parallel on-the-fly symbolic model checking
- Implementing multi-core BDD and LDD operations
- *Each DD operation low-level parallel, not just thread-safe*
- Also parallelize higher-level algorithm

Using...

- Work-stealing framework
- Scalable *lock-less* datastructures
- Parallelism using partition of the transition relation
- Custom DD operations

Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Context

Transition system

- Tuple $(S, \rightarrow)$ with states S and transitions $\rightarrow : S \times S$
- Maybe transition labels (Labeled transition system)
- Maybe state labels (Kripke structure)

Example model checking tasks

- Check some property, for example in CTL or LTL
- Count number of reachable states to satisfy our curiosity
- Find shortest path to a bad state
- Perform bisimulation minimisation

Context

Transition system

- Tuple $(S, \rightarrow)$ with states S and transitions $\rightarrow : S \times S$
- Maybe transition labels (Labeled transition system)
- Maybe state labels (Kripke structure)

Example model checking tasks

- Check some property, for example in CTL or LTL
- Count number of reachable states to satisfy our curiosity
- Find shortest path to a bad state
- Perform bisimulation minimisation

Context

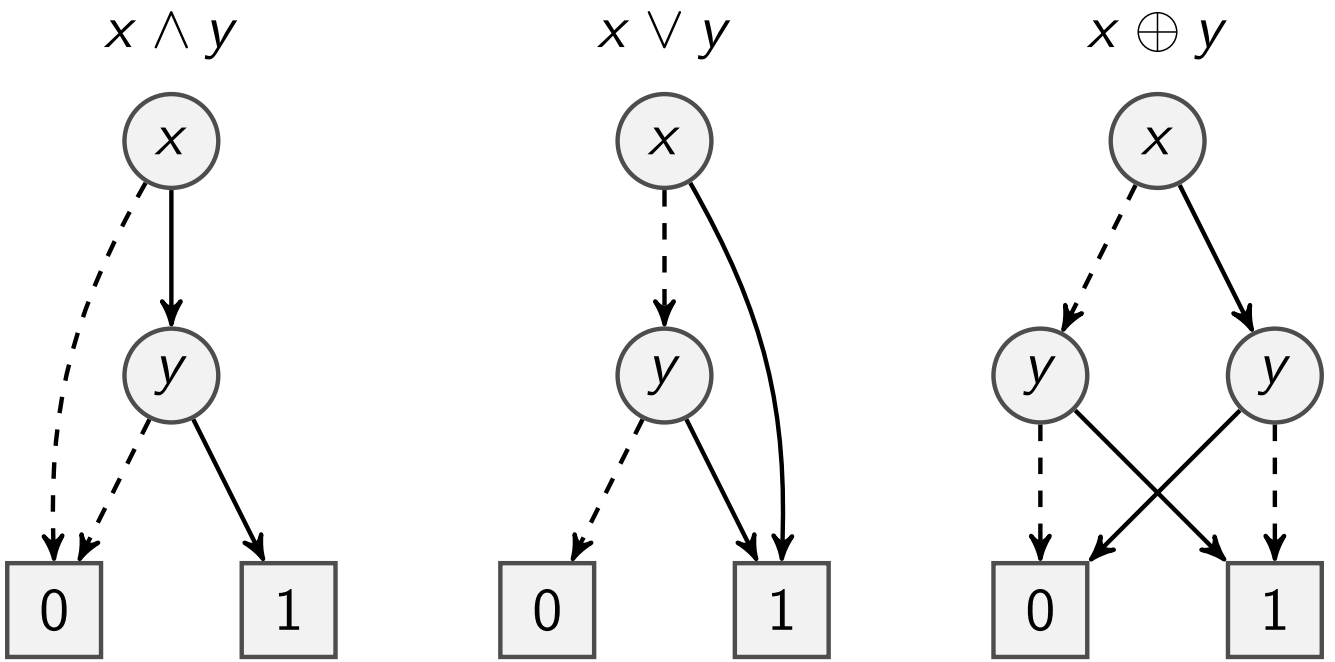
Storing sets in memory:

Explicit	Symbolic
Set of states:	Boolean function:
$\overline{x_0}\overline{x_1}\overline{x_2}$ (0,0,0)	$\overline{x_0} \vee \overline{x_1} \vee \overline{x_2}$
$\overline{x_0}\overline{x_1}x_2$ (0,0,1)	
$\overline{x_0}x_1\overline{x_2}$ (0,1,0)	All states where:
$\overline{x_0}x_1x_2$ (0,1,1)	$(0,*,*)$ or $(*,0,*)$ or $(*,*,0)$
$x_0\overline{x_1}\overline{x_2}$ (1,0,0)	
$x_0\overline{x_1}x_2$ (1,0,1)	
$x_0x_1\overline{x_2}$ (1,1,0)	

- Use symbolic methods to deal with large state spaces
- Work with characteristic function rather than individual states

Binary Decision Diagrams

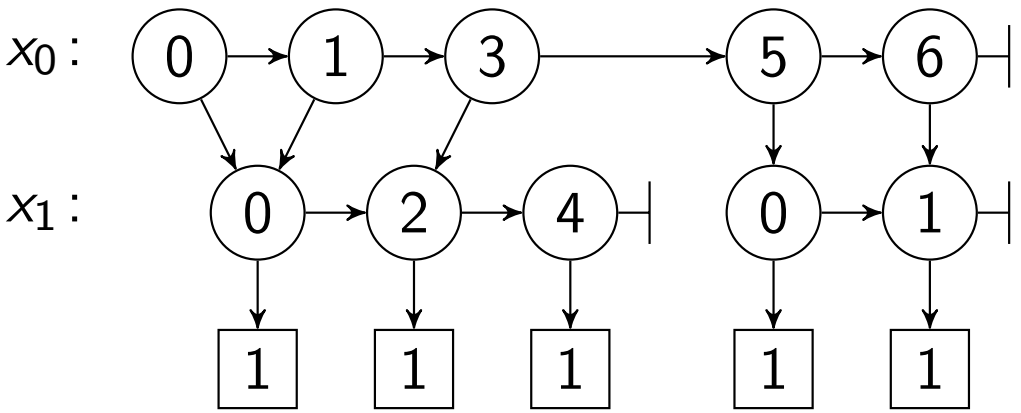
- A BDD is a directed acyclic graph encoding a $\mathbb{B}^k \to \mathbb{B}$ function
- Paths represent valuations of $\mathbb{B}^k$
- Path to $\boxed{1}$, then valuation is in the set



Context

List Decision Diagrams [Blom & Van de Pol, 2008]

- An LDD is a directed acyclic graph encoding a $\mathbb{N}^k \to \mathbb{B}$ function
- "Linked list" variation of MDDs
- LDD node: $\langle value, down, right \rangle$
- The following LDD represents the set $\{\langle 0,0\rangle, \langle 0,2\rangle, \langle 0,4\rangle, \langle 1,0\rangle, \langle 1,2\rangle, \langle 1,4\rangle, \langle 3,2\rangle, \langle 3,4\rangle, \langle 5,0\rangle, \langle 5,1\rangle, \langle 6,1\rangle\}$



Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Typical BDD operation

```
def apply(x, y, F):
    if (x, y, F) ∈ cache:
        return cache[(x, y, F)]

    if x and y are terminals:
        return F(x, y)

    v = topVar(x,y)
    low ← apply(x_{v=0}, y_{v=0}, F)
    high ← apply(x_{v=1}, y_{v=1}, F)
    result ← BDDnode(v, high, low)

    cache[(x, y, F)] ← result
    return result
```

Typical features

- Access the cache
- Use the unique table for `BDDnode`
- Two recursive calls

Task-based parallelism

Parallelizing tasks

- BDD/LDD operation is a tree of many small recursive tasks
- Work-stealing distributes tasks among workers
- Work-stealing framework: Lace (MuCoCoS 2014)
- Performance depends on properties of task tree:
  - tree size, task size, branch size, branch count, (ir)regularity

```
def apply(x, y, F):
    low ← SPAWN(apply(x_{v=0}, y_{v=0}, F))
    high ← SPAWN(apply(x_{v=1}, y_{v=1}, F))
    SYNC 2
    result ← BDDnode(v, high, low)
```

Task-based parallelism

Parallelizing tasks

- BDD/LDD operation is a tree of many small recursive tasks
- Work-stealing distributes tasks among workers
- Work-stealing framework: Lace (MuCoCoS 2014)
- Performance depends on properties of task tree:
  - tree size, task size, branch size, branch count, (ir)regularity

```
def apply(x, y, F):
    low ← SPAWN(apply(x_{v=0}, y_{v=0}, F))
    high ← SPAWN(apply(x_{v=1}, y_{v=1}, F))
    SYNC 2
    result ← BDDnode(v, high, low)
```

Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Datastructures

Safe and scalable datastructures

- BDD/LDD operations are memory-heavy:
  - Cache get & put
  - Hash table lookup (unique BDD/LDD nodes)
- Using locks/mutexes is bad for parallel performance
- *Lock-less* design using the atomic `compare_and_swap`
- Operation cache
  - Short-lived 1-bit lock on bucket during write
  - No waiting: greedy abort during "write-lock"
  - 1 `cas` instruction per write
- Unique table
  - No locks
  - 2 `cas` instructions per insert operation (if not found)

Datastructures

Safe and scalable datastructures

- BDD/LDD operations are memory-heavy:
  - Cache get & put
  - Hash table lookup (unique BDD/LDD nodes)
- Using locks/mutexes is bad for parallel performance
- *Lock-less* design using the atomic `compare_and_swap`
- Operation cache
  - Short-lived 1-bit lock on bucket during write
  - No waiting: greedy abort during "write-lock"
  - 1 `cas` instruction per write
- Unique table
  - No locks
  - 2 `cas` instructions per insert operation (if not found)

Datastructures

Safe and scalable datastructures

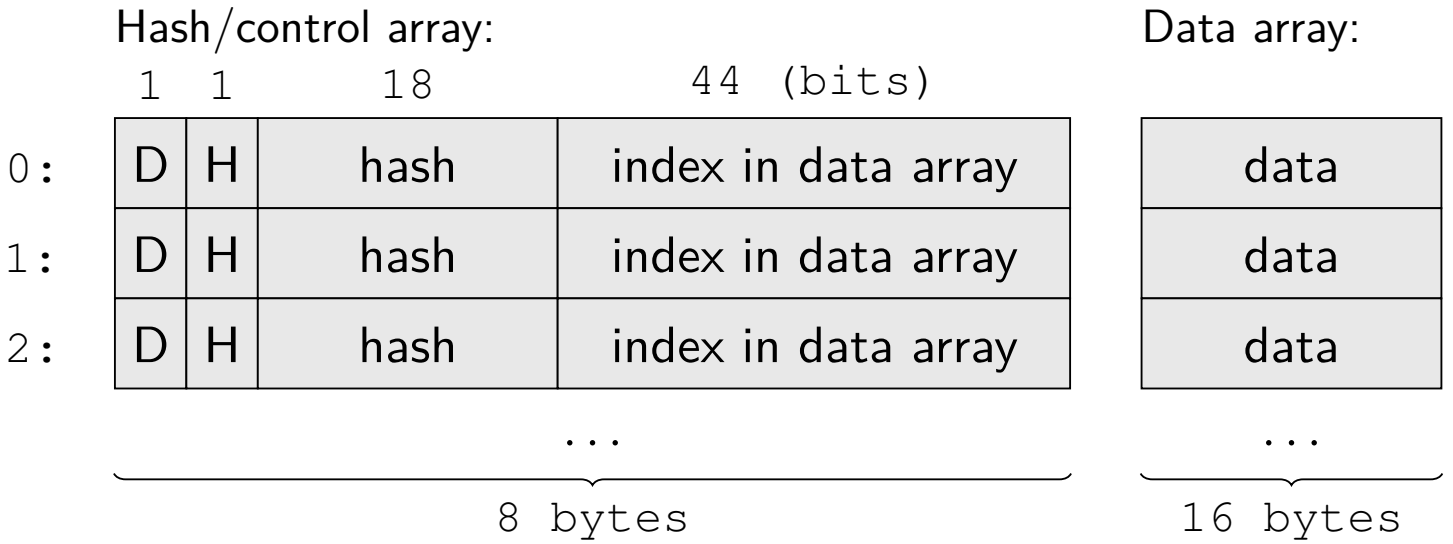
- BDD/LDD operations are memory-heavy:
  - Cache get & put
  - Hash table lookup (unique BDD/LDD nodes)
- Using locks/mutexes is bad for parallel performance
- *Lock-less* design using the atomic `compare_and_swap`
- Operation cache
  - Short-lived 1-bit lock on bucket during write
  - No waiting: greedy abort during "write-lock"
  - 1 `cas` instruction per write
- Unique table
  - No locks
  - 2 `cas` instructions per insert operation (if not found)

Datastructures

Safe and scalable datastructures

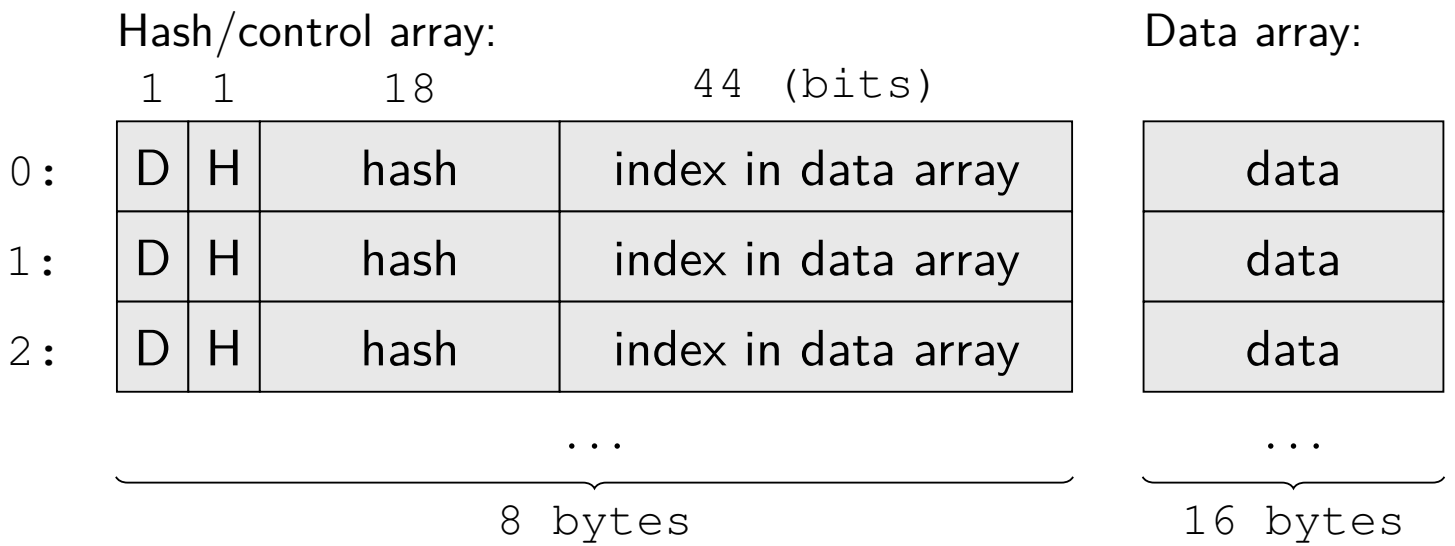
- BDD/LDD operations are memory-heavy:
  - Cache get & put
  - Hash table lookup (unique BDD/LDD nodes)
- Using locks/mutexes is bad for parallel performance
- *Lock-less* design using the atomic `compare_and_swap`
- Operation cache
  - Short-lived 1-bit lock on bucket during write
  - No waiting: greedy abort during "write-lock"
  - 1 `cas` instruction per write
- Unique table
  - No locks
  - 2 `cas` instructions per insert operation (if not found)

Unique table



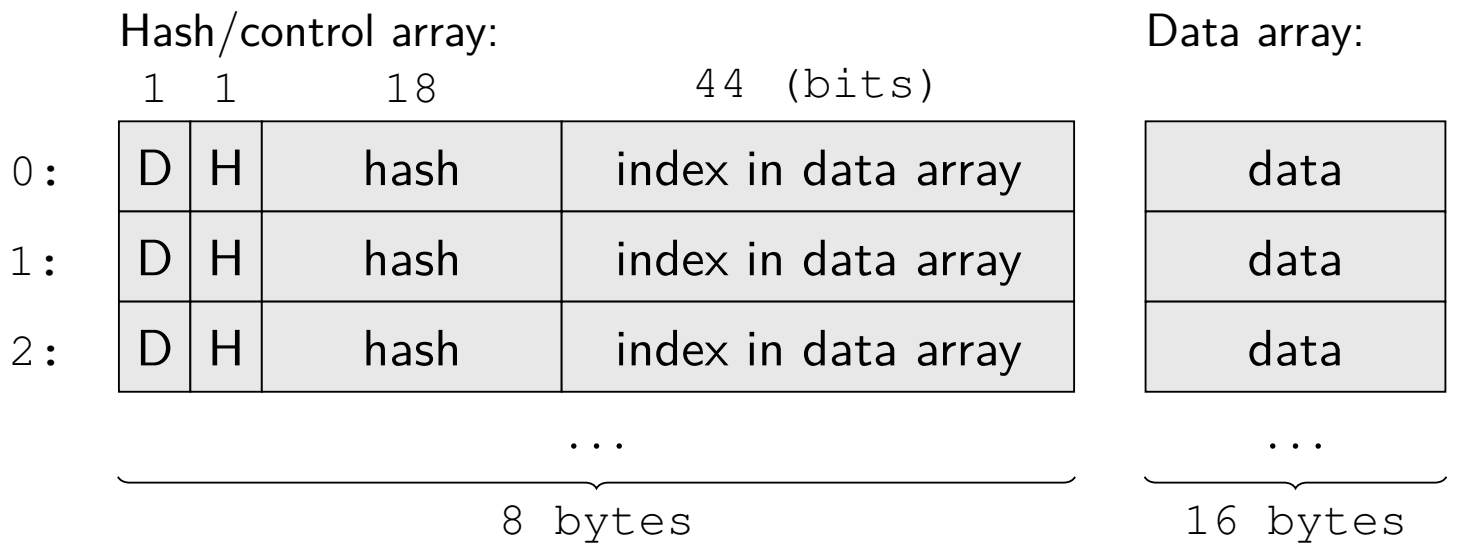
- D: bucket in data array used (field “data”)
- H: bucket in hash array used (fields “hash” and “index”)
- Manipulate using `compare_and_swap`

Unique table



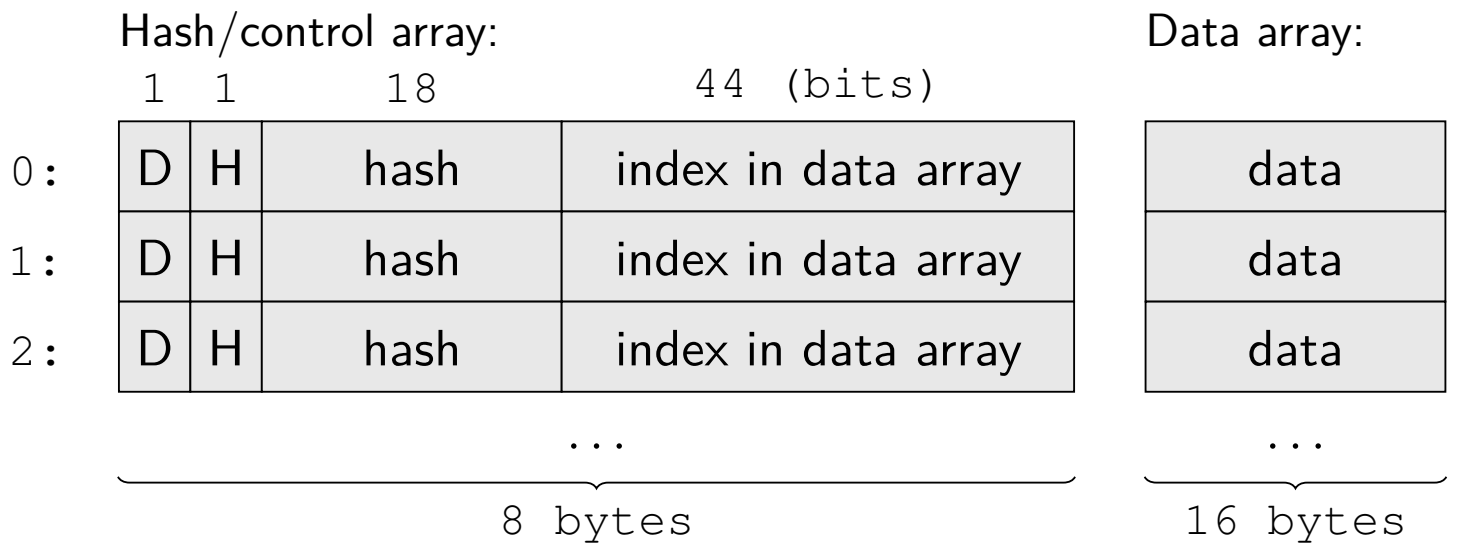
- Phase 1: find empty hash bucket or existing entry
  - Using cache-line aware version of linear probing
  - On empty hash bucket: go to Phase 2
- Phase 2: find empty data bucket (linear search)
  - On empty data bucket: use `cas` to set "D"
- Phase 3: find empty hash bucket or existing entry
  - Continue search from Phase 1
  - On empty hash bucket: use `cas` to set "H", "hash" and "index"
  - On existing: undo setting "D" in phase 2 using `cas`

Unique table



- Phase 1: find empty hash bucket or existing entry
  - Using cache-line aware version of linear probing
  - On empty hash bucket: go to Phase 2
- Phase 2: find empty data bucket (linear search)
  - On empty data bucket: use `cas` to set "D"
- Phase 3: find empty hash bucket or existing entry
  - Continue search from Phase 1
  - On empty hash bucket: use `cas` to set "H", "hash" and "index"
  - On existing: undo setting "D" in phase 2 using `cas`

Unique table



- Phase 1: find empty hash bucket or existing entry
  - Using cache-line aware version of linear probing
  - On empty hash bucket: go to Phase 2
- Phase 2: find empty data bucket (linear search)
  - On empty data bucket: use `cas` to set "D"
- Phase 3: find empty hash bucket or existing entry
  - Continue search from Phase 1
  - On empty hash bucket: use `cas` to set "H", "hash" and "index"
  - On existing: undo setting "D" in phase 2 using `cas`

Cache

- Stores 4 DD + operation identifier $\mapsto$ 64-bit result
- 32 bytes per cache bucket
- Writing to the cache with `cas`-lock: lock, write, release
- Lossy (overwriting) cache + abort read/write when locked
- Losing results is fine!

Garbage collection

Parallel garbage collection

- Stop the world (all workers do garbage collection)
- Triggered when the unique table is "full"

Modular garbage collection

- Clear hash table (hash/control array)
- Call all marking handlers (mark for rehash)
  - Internal list of references during operations
  - Custom user reference management
- Optional: resize tables
- Rehash all marked nodes

Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Computing reachability

- Compute reflexive transitive closure of $\mathcal{T}$ applied to $\mathcal{S}$
- Use a frontier set (only new states)

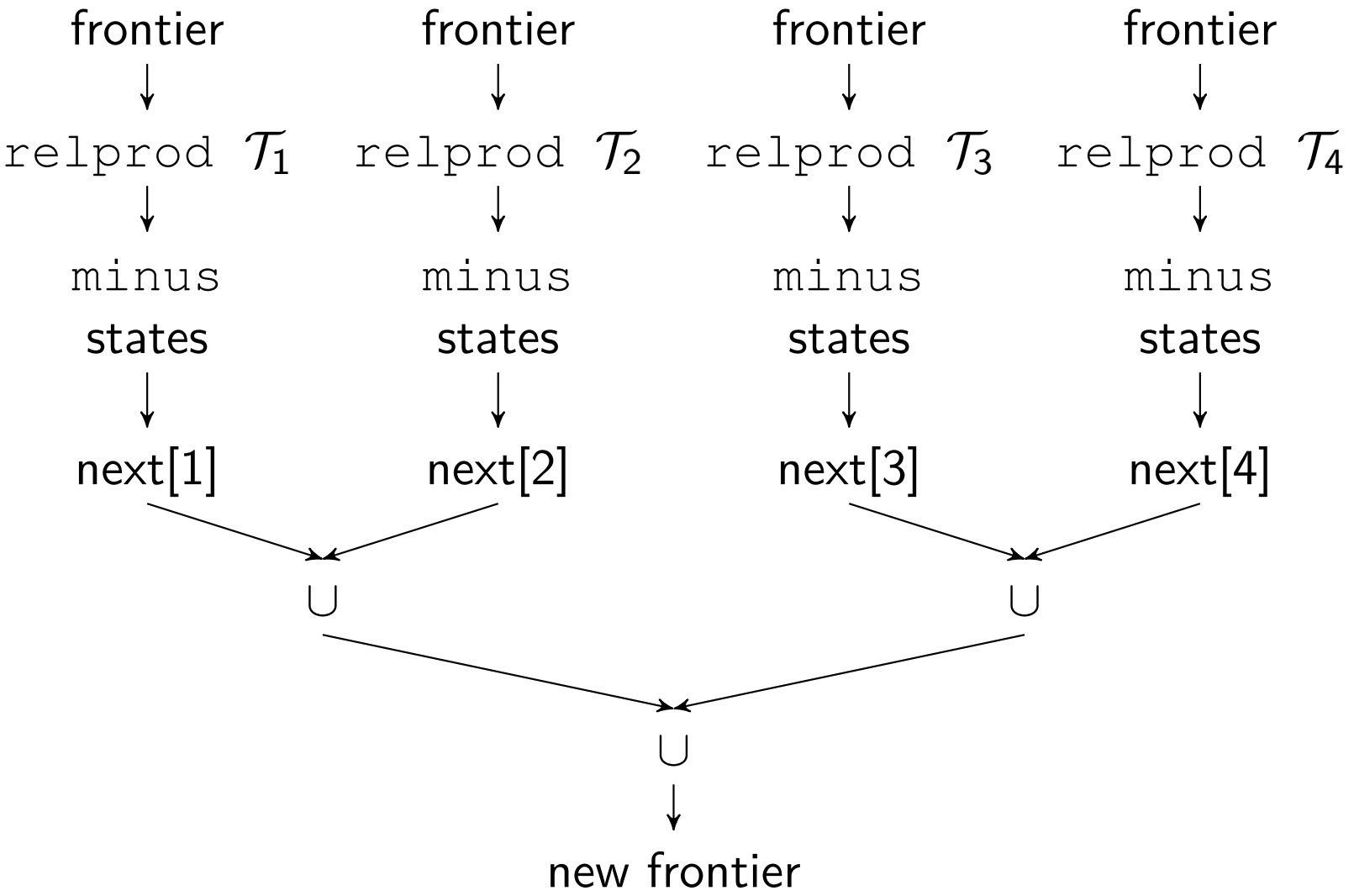
```
def ClosureFS(S, T):
    states ← ∅
    frontier ← S
    while frontier ≠ ∅:
        next ← RelProd(frontier, T)
        frontier ← next \ states
        states ← states ∪ frontier
    return states
```

Computing reachability

- Compute reflexive transitive closure of $\mathcal{T}$ applied to $\mathcal{S}$
- Multiple transition relations

```
def ClosureFS(S, T_1, ..., T_N):
    states ← ∅
    frontier ← S
    while frontier ≠ ∅:
        foreach T_i do
            next[i] ← RelProd(frontier, T_i)
            next[i] ← next[i] \ states
        end
        frontier = next[1] ∪ ... ∪ next[N]
        states ← states ∪ frontier
    return states
```

Parallel reachability



Parallel reachability

```
def par_closure(frontier, states, T_1, ..., T_N, i, n):
    if n = 1:
        return RelProd(frontier, T_i) \ states
    else:
        left ← SPAWN(par_closure(..., i, n/2))
        right ← SPAWN(par_closure(..., i + n/2, n − n/2))
        SYNC 2
        return left ∪ right
```

Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Computing reachability

- Compute reflexive transitive closure of $\mathcal{T}$ applied to $\mathcal{S}$
- Multiple transition relations
- On-the-fly learning via (explicit) next-state interface

def ClosureFS($\mathcal{S}, \mathcal{T}_1, \ldots, \mathcal{T}_N$)**:**
    states $\leftarrow \emptyset$
    frontier $\leftarrow \mathcal{S}$
    while frontier $\neq \emptyset$**:**
        foreach $\mathcal{T}_i$ **do**
            $\mathcal{T}_i \leftarrow$ Learn(frontier, $\mathcal{T}_i$)
            next[i] $\leftarrow$ RelProd(frontier, $\mathcal{T}_i$)
            next[i] $\leftarrow$ next[i] $\setminus$ states
        end
        frontier $=$ next[1] $\cup \ldots \cup$ next[N]
        states $\leftarrow$ states $\cup$ frontier
    return states

Parallel learning

Algorithm overview

- For every (projected) state in the frontier set, ask successors
- Add all successors to transition relation

Implementation

- Custom BDD/LDD operation collect to integrate 2 functions:
  1. enumerate all states using callback function
  2. union the results of callback function

Parallel learning

Algorithm overview

- For every (projected) state in the frontier set, ask successors
- Add all successors to transition relation

Implementation

- Custom BDD/LDD operation collect to integrate 2 functions:
  1. enumerate all states using callback function
  2. union the results of callback function

Parallel learning

```
def ldd_collect(S, state, NEXT):
    if S is terminal 1:
        return NEXT(state)
    elif S is terminal 0:
        return ∅
    else:
        right ← SPAWN(ldd_collect(S.right, state, NEXT))
        down ← SPAWN(ldd_collect(S.down, state + [S.value], NEXT))
        SYNC 2
        return down ∪ right

def Learn(S, T, NEXT):
    return T ∪ ldd_collect(S, ∅, NEXT)
```

Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Results

Benchmarks

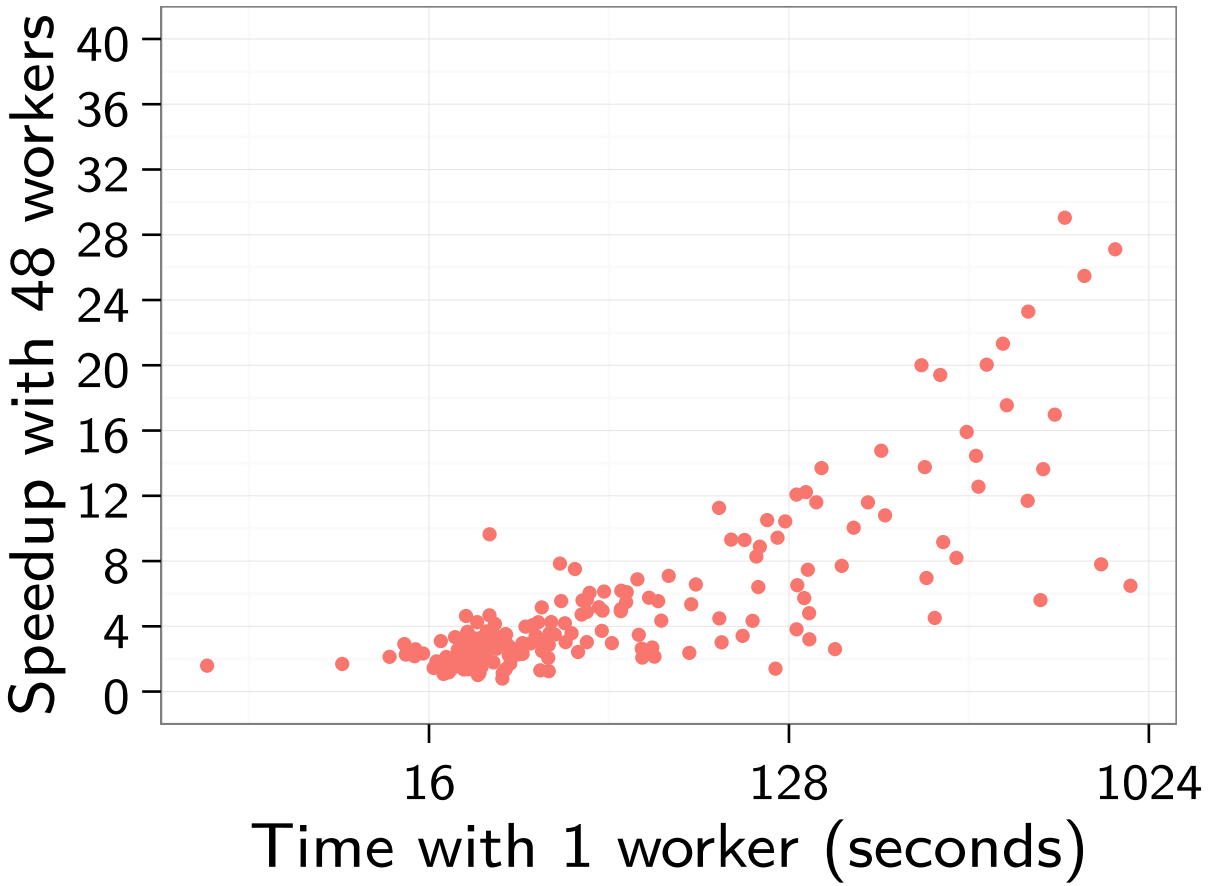
- Machine: 48-core Opteron 6168 (1.9GHz), 128GB memory.
- Experiments using entire BEEM database (276 DiVinE models)
  - 7 timed out (1200 seconds)
  - Each data point repeated at least 3 times
- LTSmin & LDD in Sylvan
- Variants:
  - bfs: only low-level parallel
  - par also high-level parallel and parallel learning

Results

Experiment	T_1	T_{16}	T_{24}	T_{32}	T_{48}	**Speedup** T_{48}/T_1
`blocks.4` (bfs)	630.04	45.88	33.24	27.01	21.69	**29.0**
`blocks.4` (par)	629.54	41.61	29.26	23.04	16.58	**38.0**
`telephony.8` (bfs)	843.06	66.28	47.91	39.17	31.10	**27.1**
`telephony.8` (par)	843.70	58.17	41.17	32.76	24.68	**34.2**
`lifts.8` (bfs)	377.36	36.61	30.06	27.68	26.11	**14.5**
`lifts.8` (par)	377.52	25.92	18.68	15.18	12.03	**31.4**
`firewire_tree.1` (bfs)	16.43	11.24	11.12	11.36	11.35	**1.4**
`firewire_tree.1` (par)	16.40	1.09	0.97	0.94	0.99	**16.5**
Sum of all bfs	20745	3902	3667	3625	3737	**5.6**
Sum of all par	20756	1851	1552	1403	1298	**16.0**

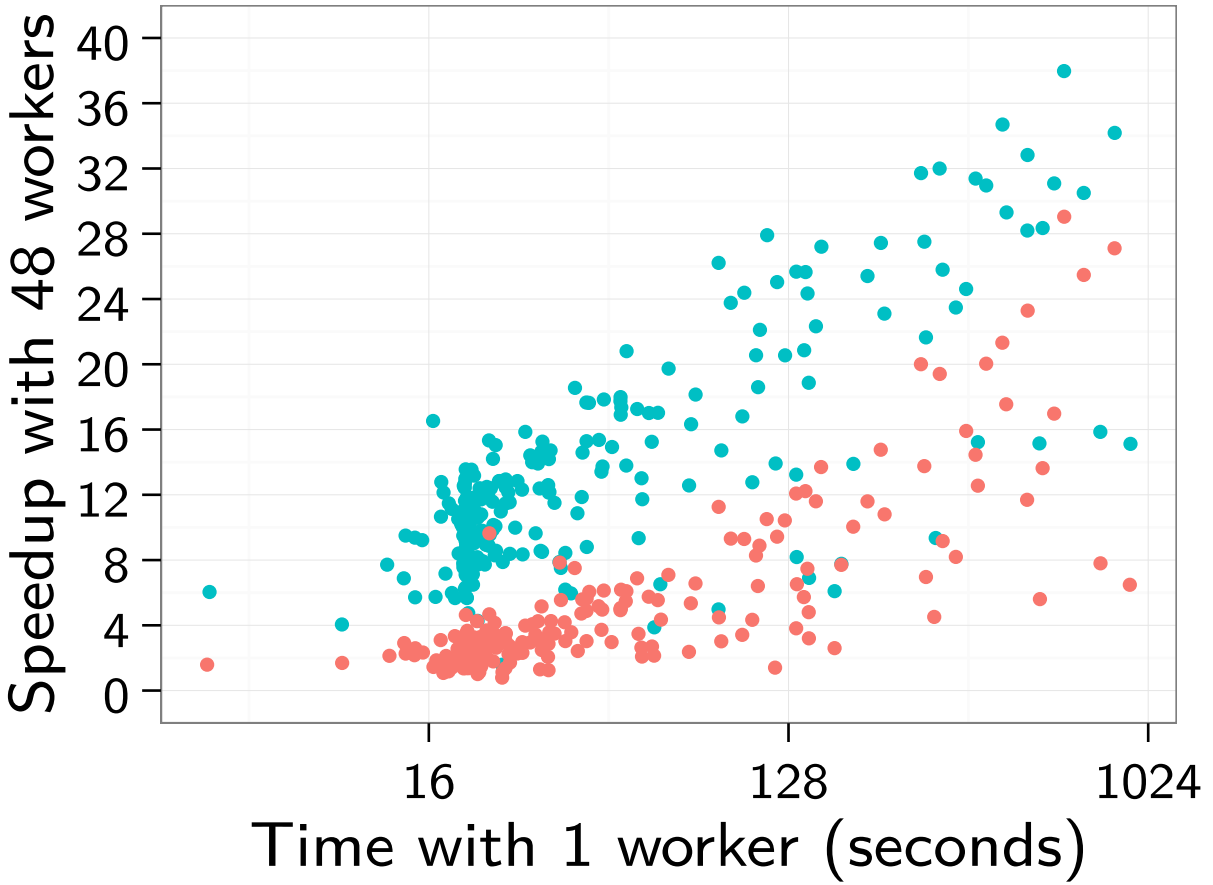
Results

Only low-level parallelism:

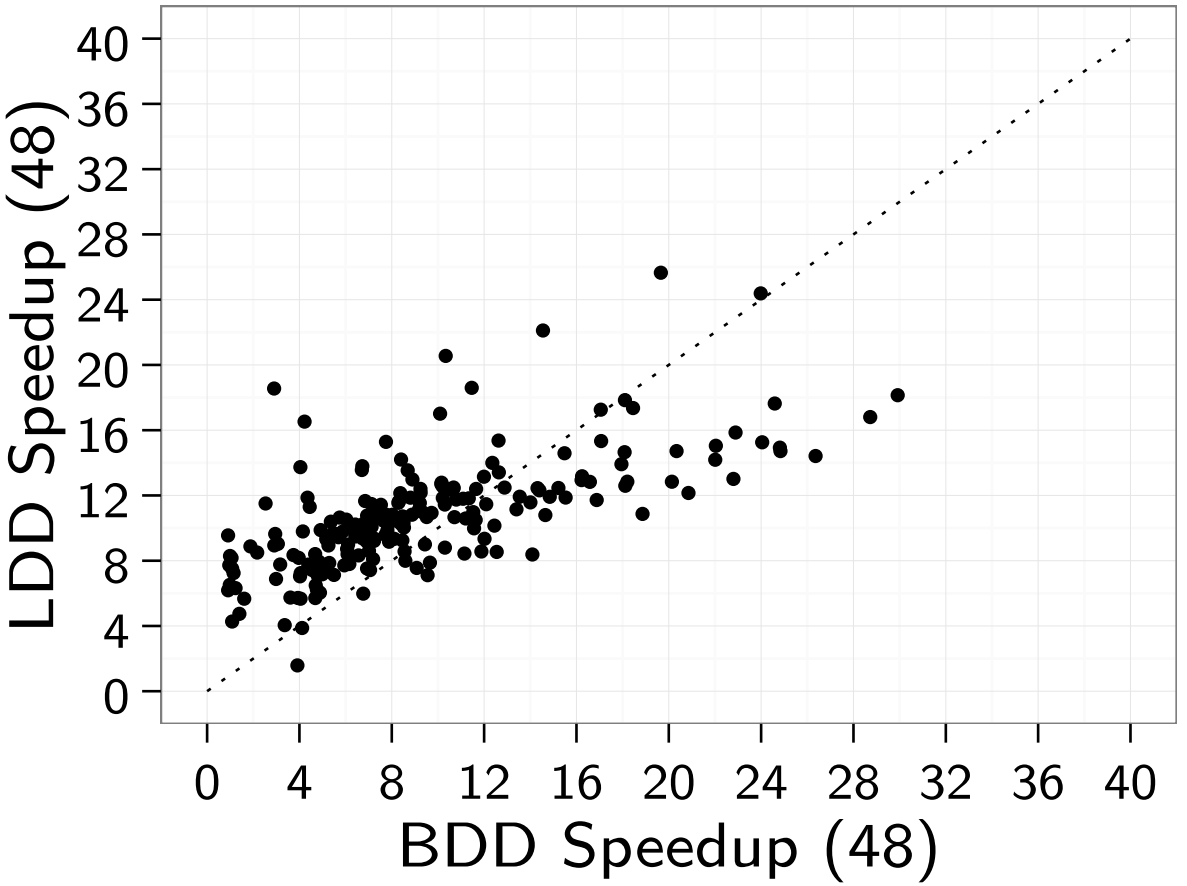


Results

Low-level parallelism plus parallel learning and parallel calls



Results



Outline

Context

Low-level parallelism
- Parallelizing BDD/LDD operations
- Efficient parallel datastructures

Higher level parallelism
- Multiple transition relations
- Extending Sylvan for parallel learning

Experimental evaluation

Conclusions

Extending Sylvan

Extending Sylvan is easy!

- Sylvan framework provides:
  - Easy access to cache and unique table
  - Reuse internal referencing for garbage collection
  - Integration with Lace work-stealing framework
- Easy to add other types of decision diagrams
- Promising research: custom multi-core BDD operations for large gains in symbolic bisimulation minimisation
  - Over 250x faster branching bisimulation minimisation
  - Compared to Sigref using standard CuDD operations
- Bindings for Java, Python and Haskell exist

Extending Sylvan

Extending Sylvan is easy!

- Sylvan framework provides:
  - Easy access to cache and unique table
  - Reuse internal referencing for garbage collection
  - Integration with Lace work-stealing framework
- Easy to add other types of decision diagrams
- Promising research: custom multi-core BDD operations for large gains in symbolic bisimulation minimisation
  - Over 250x faster branching bisimulation minimisation
  - Compared to Sigref using standard CuDD operations
- Bindings for Java, Python and Haskell exist

Extending Sylvan

Extending Sylvan is easy!

- Sylvan framework provides:
  - Easy access to cache and unique table
  - Reuse internal referencing for garbage collection
  - Integration with Lace work-stealing framework
- Easy to add other types of decision diagrams
- Promising research: custom multi-core BDD operations for large gains in symbolic bisimulation minimisation
  - Over 250x faster branching bisimulation minimisation
  - Compared to Sigref using standard CuDD operations
- Bindings for Java, Python and Haskell exist

Extending Sylvan

Extending Sylvan is easy!

- Sylvan framework provides:
  - Easy access to cache and unique table
  - Reuse internal referencing for garbage collection
  - Integration with Lace work-stealing framework
- Easy to add other types of decision diagrams
- Promising research: custom multi-core BDD operations for large gains in symbolic bisimulation minimisation
  - Over 250x faster branching bisimulation minimisation
  - Compared to Sigref using standard CuDD operations
- Bindings for Java, Python and Haskell exist

Conclusions

Contributions

- Sylvan, a scalable parallel BDD+LDD library
- Easy to extend with other DD types and new operations
- Parallel reachability strategy using disjunctive partitioning
- Parallel on-the-fly transition learning
- `https://github.com/utwente-fmt/sylvan`

Future Work

- Symbolic bisimulation minimisation (for LTS and IMC)
- MTBDDs, MTLDDs, HSDDs, EVMDDs, ZDDs, etc
- Dynamic variable reordering; parallel saturation
- Other applications, e.g. synchronous models/hardware

Backup slides

Backup slides

Reachability

Explicit-state reachability

```
def reachable(initial, trans):
    visited = ∅
    stack.push(initial)
    while len(stack) > 0:
        state = stack.pop()
        visited = visited ∪ state
        for s in [s in trans(state) if s ∉ visited]:
            stack.push(s)
    return visited
```

Symbolic reachability

```
def reachable(initial, trans):
    states = {initial}
    prev = ∅
    while prev != states:
        prev = states
        next = relprods(states, trans)
        states = union(states, next)
    return states
```

Symbolic reachability

Algorithm 1 (bfs)

```
def reachable(initial, trans):
    states = {initial}
    prev = ∅
    while prev != states:
        prev = states
        next = relprods(states, trans)
        states = union(states, next)
    return states
```

Algorithm 2 (bfs–prev)

```
def reachable(initial, trans):
    states = {initial}
    prev = states
    while prev != ∅:
        next = relprod(prev, trans)
        prev = minus(next, states)
        states = union(states, next)
    return states
```

Models

Model	Depth	Groups	States	BDD Nodes	Work
bakery.4	103	21	157,003	12,009	6,003,764
schedule_world.2	16	26	1,570,340	18,779	18,376,072
lifts.4	115	69	112,792	118,212	44,359,194
peterson.6	79	20	174,495,861	38,169	57,614,947
lifts.5	115	69	191,567	173,495	88,308,709
iprotocol.4	166	26	3,290,916	113,774	97,854,082
leader_election.5	159	88	4,803,952	62,682	109,241,755
lifts.6	214	91	333,649	392,470	115,905,084
bakery.6	129	28	11,845,035	148,119	130,645,221
bakery.5	288	28	7,866,401	156,771	198,664,965
leader_filters.7	70	24	26,302,351	326,990	203,210,754
leader_election.6	220	99	35,777,100	106,124	250,620,273
schedule_world.3	21	34	166,649,331	28,500	265,204,724
leader_filters.6	84	20	220,913,716	466,178	286,035,694
peterson.7	175	25	142,471,098	72,275	286,336,555
iprotocol.6	452	26	41,387,484	190,427	314,269,205
lifts.7	218	91	5,126,781	569,314	573,845,199
iprotocol.5	232	26	31,071,582	545,578	665,267,682
iprotocol.7	279	26	59,794,192	676,092	1,000,748,636
bakery.7	448	28	29,047,471	681,012	1,018,315,634
bakery.8	293	35	253,131,202	582,463	1,452,973,233
collision.5	179	29	431,965,993	29,537	1,747,001,660

Related work

Dynamic programming problems can be parallelized using fine-grained task-based parallelism (e.g. work-stealing).

Frameworks: Cilk [Blumofe et al, 1995] and Wool [Faxén 2008]

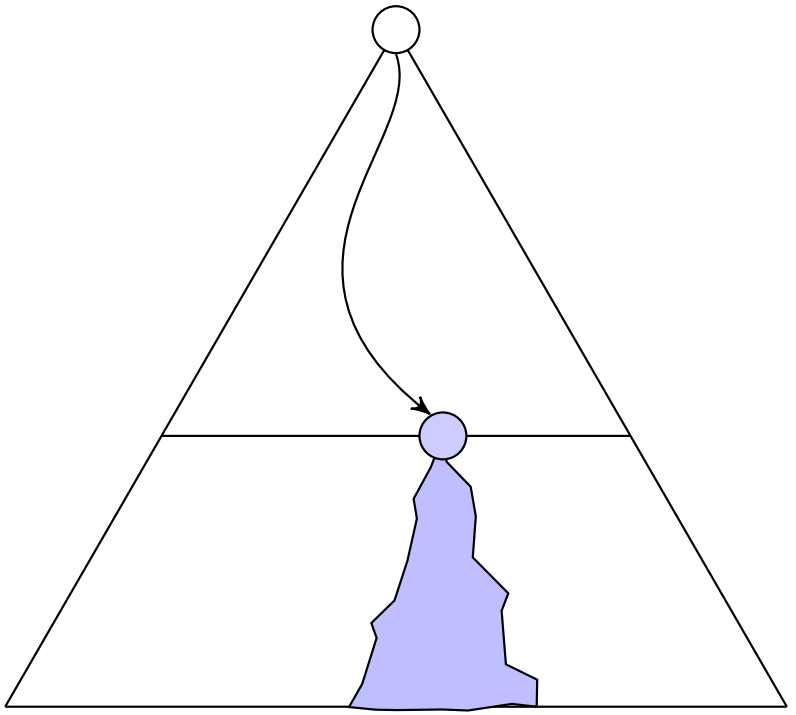
Parallelization using slicing [Grumberg et al. 2005, 2006]

Parallel saturation (with Cilk) [Ezekiel et al. CAV 2007]

Parallel saturation based model checking [Vörös et al. ISPDC 2011]

Symbolic representation

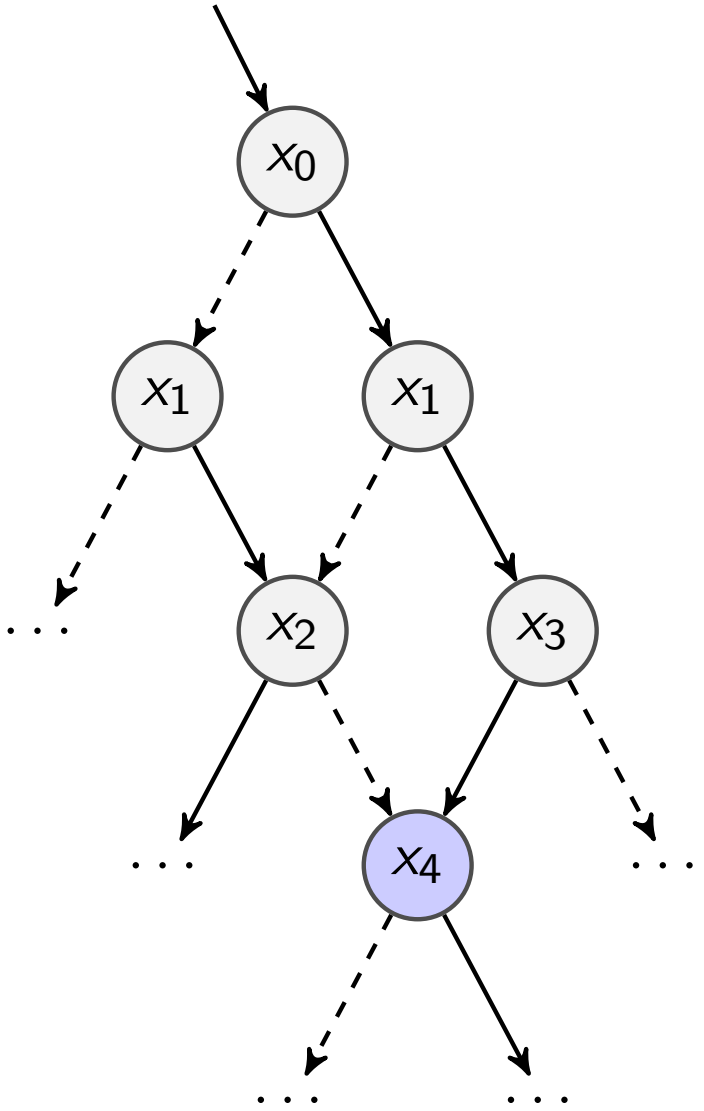
Symbolic methods use binary decision diagrams. Example: function that maps $\vec{x}$ to some function g, $f(x) = g$, with all variables of $\vec{x}$ ordered before variables in g.



path σ representing valuations of $\vec{x}$

subgraph of f(x), for each $x \in \sigma$

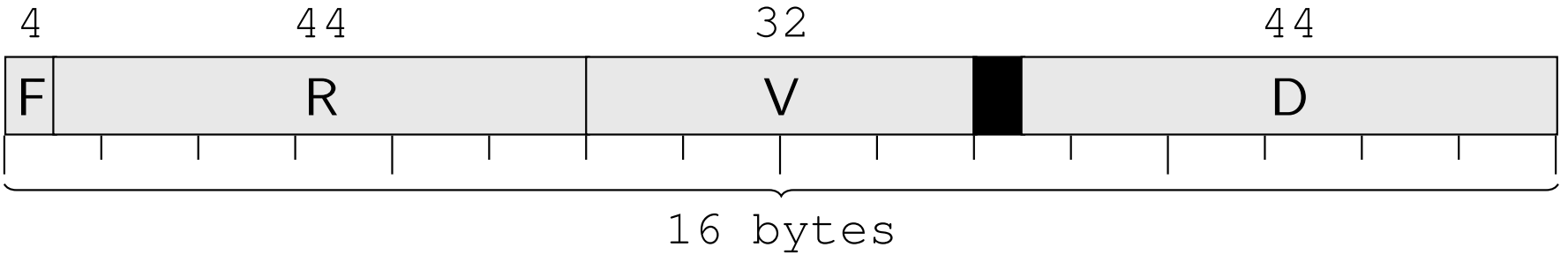
Typical BDD operation



Caching

- Different paths to ○ for different valuations of $x_0, \ldots, x_3$.
- Operations on ○ typically have the same result
- Cache is necessary
- Cache can be lossy

LDD node layout



Each LDD node is 16 bytes

- F: flags (for special nodes/extensions)
- R: right LDD
- V: value
- D: down LDD

Memory requirements

Memory usage

- 36 bytes per cache bucket
  - 4 bytes overhead per 32 bytes data (+12.5%)
- 24 bytes per BDD/LDD node (table)
  - 8 bytes overhead per 16 bytes data (+50%)
- Some overhead from program stack and work stealing

Table resizing

- Pre-allocate maximum size table in **virtual** memory
- Only use a part of the table
- Kernel allocates page in **real** memory