Techniques for Memory-Efficient Model Checking of C and C++ Code

Petr Ročkai **Vladimír Štill** Jiří Barnat

ParaDiSe
Parallel & Distributed Systems Laboratory

Masaryk University

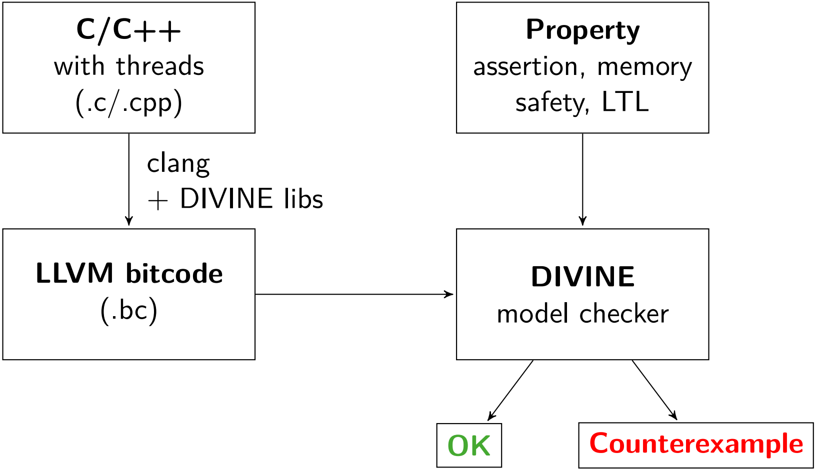
Brno, Czech Republic

SEFM 2015

DIVINE

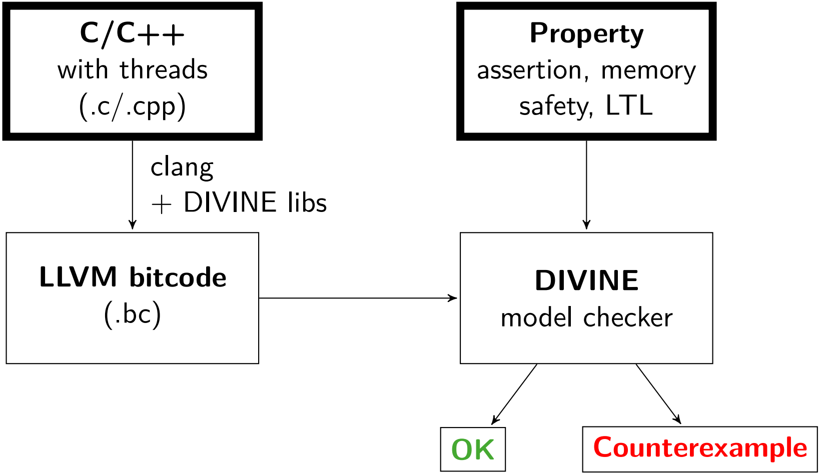
What we do

- verification of C & C++ programs
- using LLVM bitcode
- support for threads, using `pthreads` or C++ standard threads
- support for large parts of C & C++ library

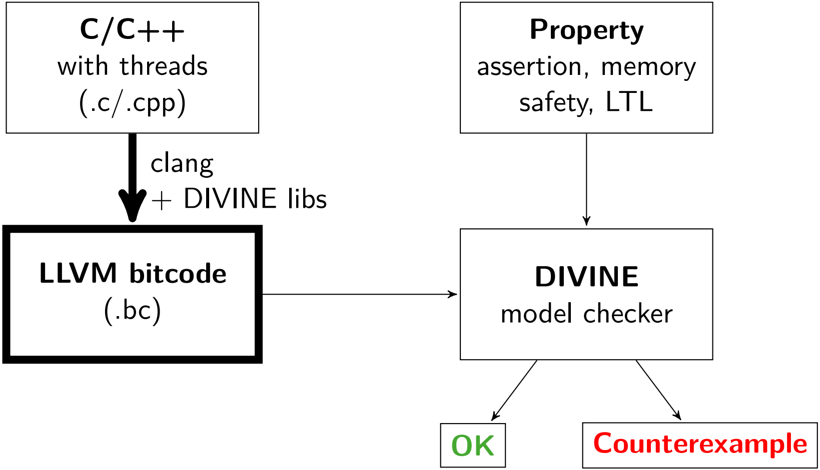


model checking programs with DIVINE

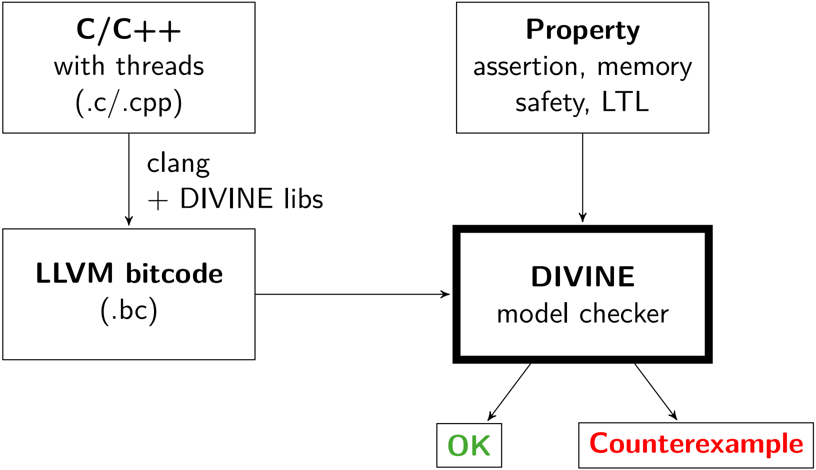
Introduction



programmer gives inputs: source code and specification

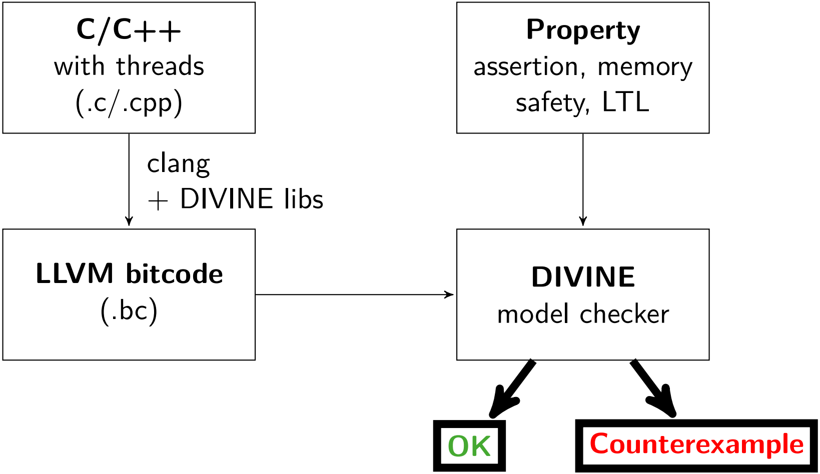


the program is compiled into LLVM bitcode

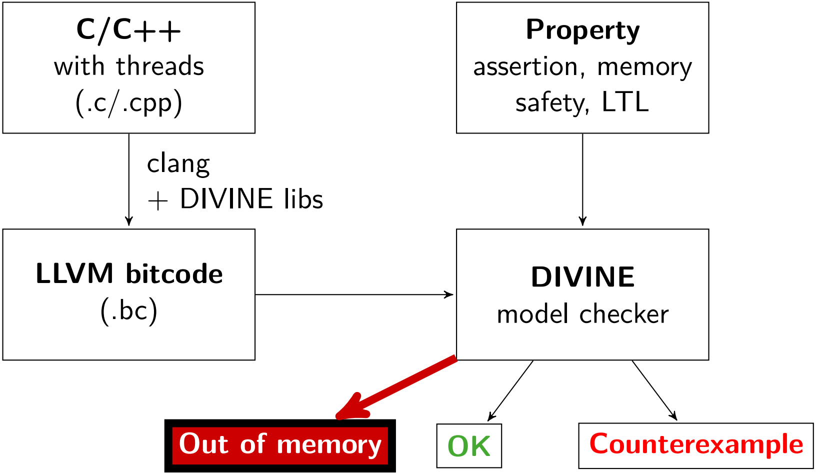


DIVINE explores all relevant interleavings

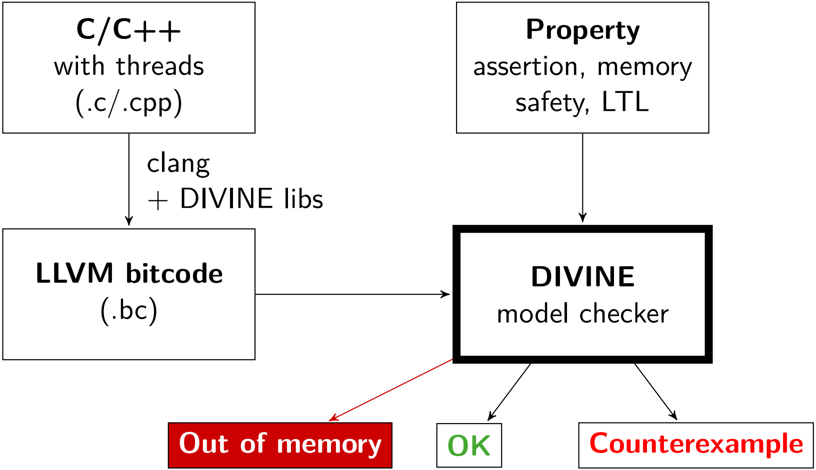
Introduction



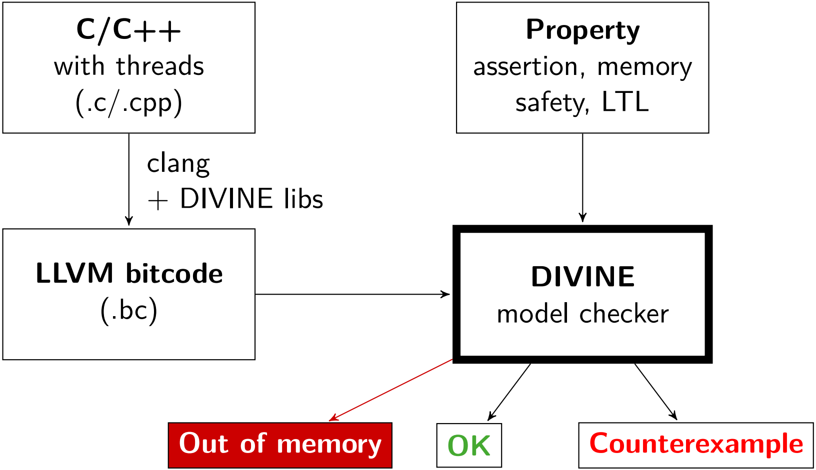
verification results



verification is memory and time consuming



how to optimize model-checker's memory consumption?



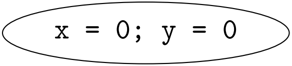
need to know how it works

Explicit-State Model Checking

explores all relevant outcomes of program:

explores all relevant outcomes of program:

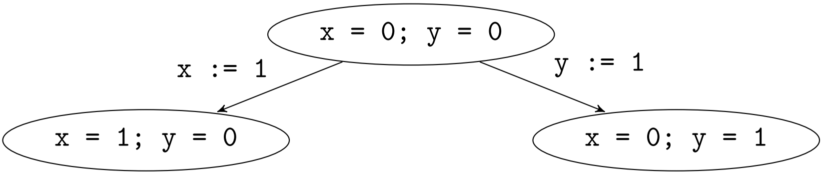
- starts from an initial state



Explicit-State Model Checking

explores all relevant outcomes of program:

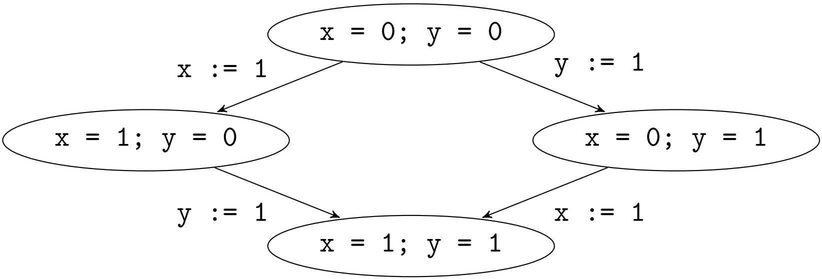
- starts from an initial state
- looks at possible actions that can be taken in each state



Explicit-State Model Checking

explores all relevant outcomes of program:

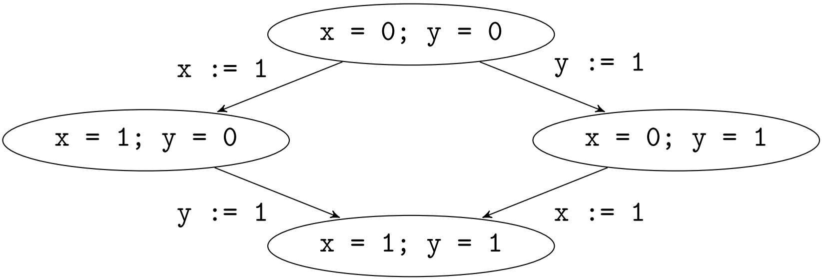
- starts from an initial state
- looks at possible actions that can be taken in each state



Explicit-State Model Checking

explores all relevant outcomes of program:

- starts from an initial state
- looks at possible actions that can be taken in each state

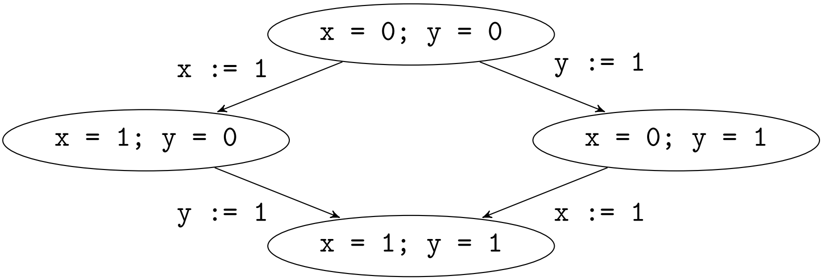


- builds state space

Explicit-State Model Checking

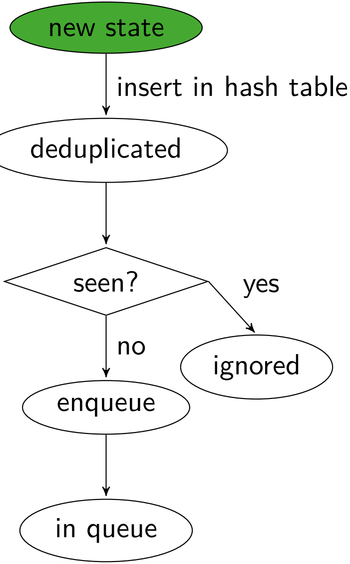
explores all relevant outcomes of program:

- starts from an initial state
- looks at possible actions that can be taken in each state



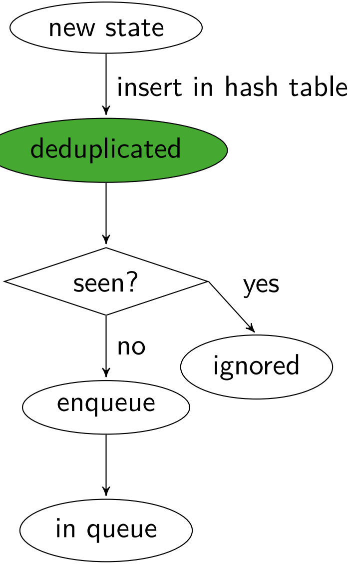
- builds state space
- graph exploration

State Processing in DIVINE



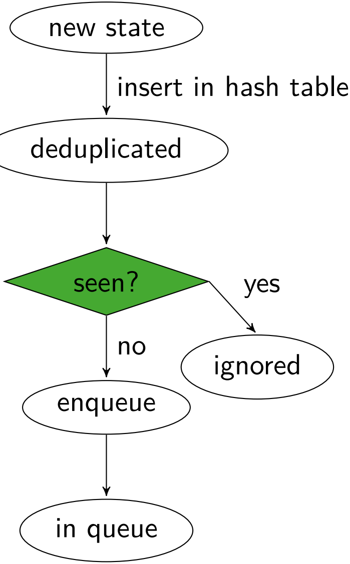
new state

- generated by state space generator
- allocated as a linear block of new memory
- same state (content-wise) can exist in hash table



deduplicated

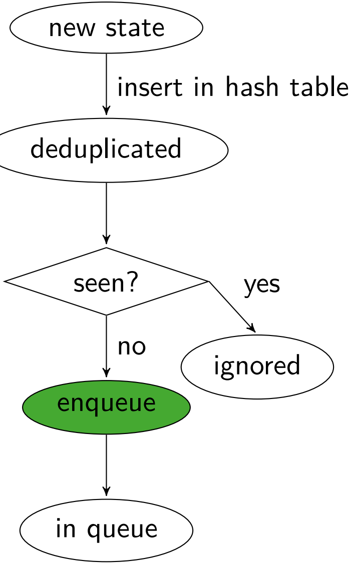
- attempt insertion into the hash table
- if already present, deallocate the new state
- proceed using the state stored in the hash table
- hash table contains pointer to the state memory



seen?

- algorithm decides how to process the state
- detect property violation

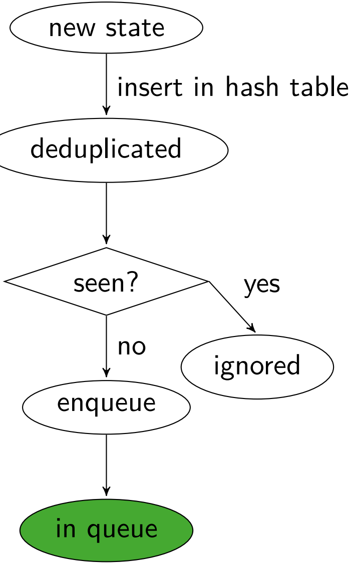
State Processing in DIVINE



enqueue

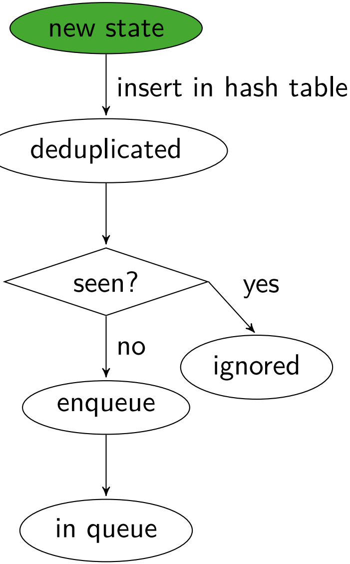
- push the state into the exploration queue

State Processing in DIVINE



in queue

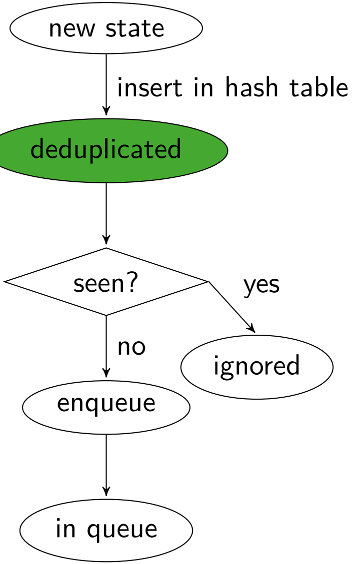
- the queue contains pointers (to the same memory location as the hash table does)



observations

- a state is allocated in **new state**

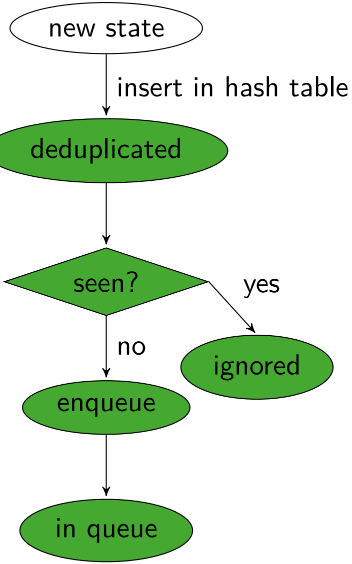
State Processing in DIVINE



observations

- a state is allocated in **new state**
- and deallocated if it is a duplicate

State Processing in DIVINE



observations

- a state is allocated in **new state**
- and deallocated if it is a duplicate
- most of the time, only one copy of the state exists
- a pointer to this canonic copy is stored in the hash table.

Hash Table

- why a hash table?
  - fast insert and lookup
  - simple
  - memory efficient

Hash Table

- why a hash table?
  - fast insert and lookup
  - simple
  - memory efficient
- stored states take up almost all memory
  - individual states are large
  - and often similar
  - → unnecessary redundancy

Hash Table

- why a hash table?
  - fast insert and lookup
  - simple
  - memory efficient
- stored states take up almost all memory
  - individual states are large
  - and often similar
  - → unnecessary redundancy
- we need a **compressed data structure** with behaviour similar to a hash table
  - associative container
  - capable of storing variable length keys
  - can grow

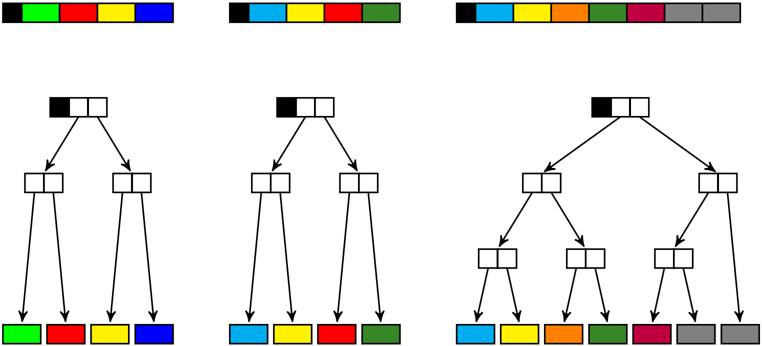
Tree Compression

uses redundancy between states:

original states (black = associated data)

uses redundancy between states:

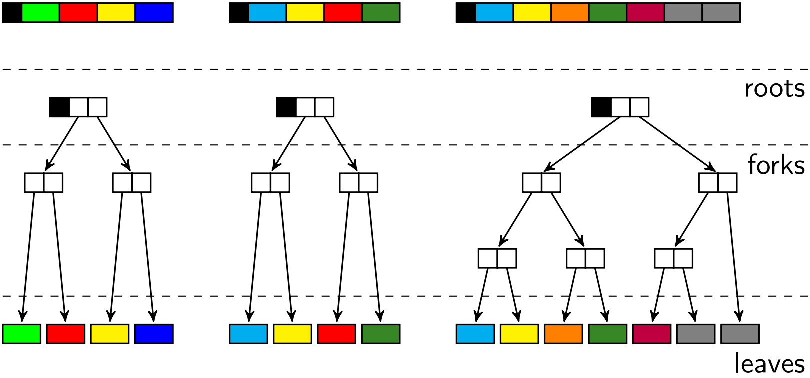
original states + tree decomposition



Tree Compression

uses redundancy between states:

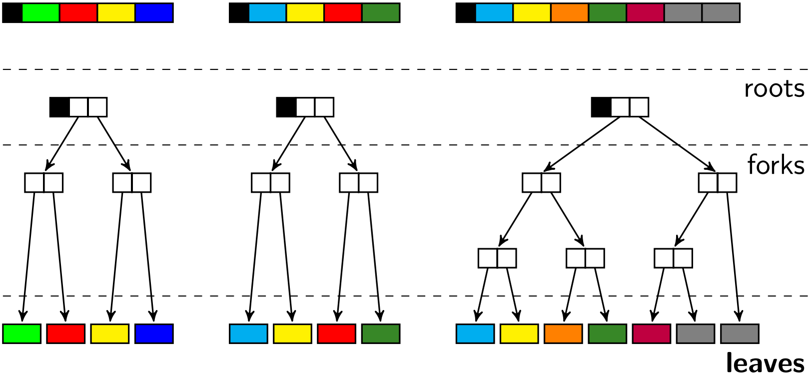
three types of tree nodes



Tree Compression

uses redundancy between states:

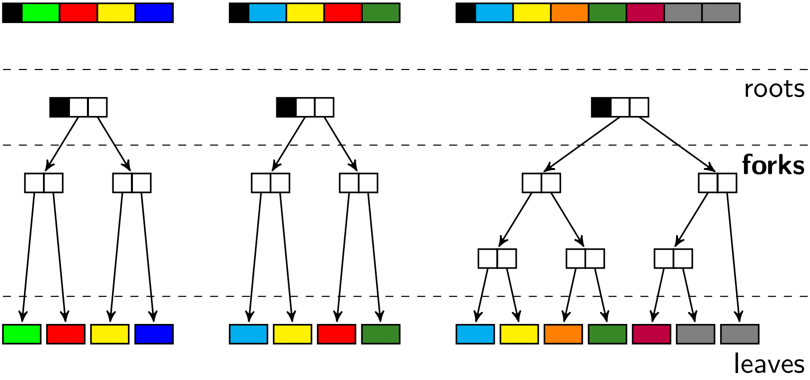
three types of tree nodes: **leaves** – parts of original state



Tree Compression

uses redundancy between states:

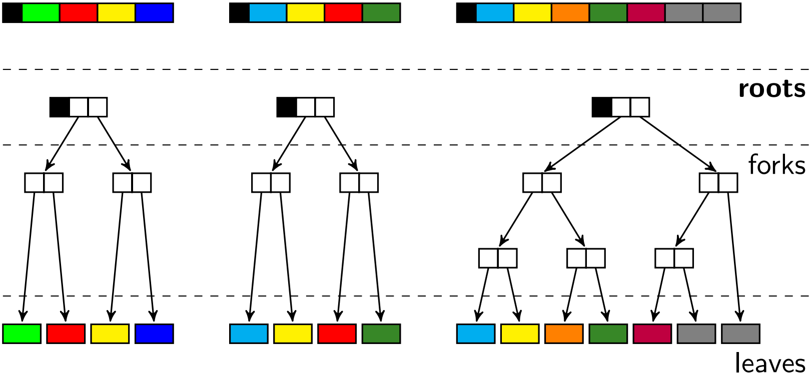
three types of tree nodes: forks – connect larger parts of state



Tree Compression

uses redundancy between states:

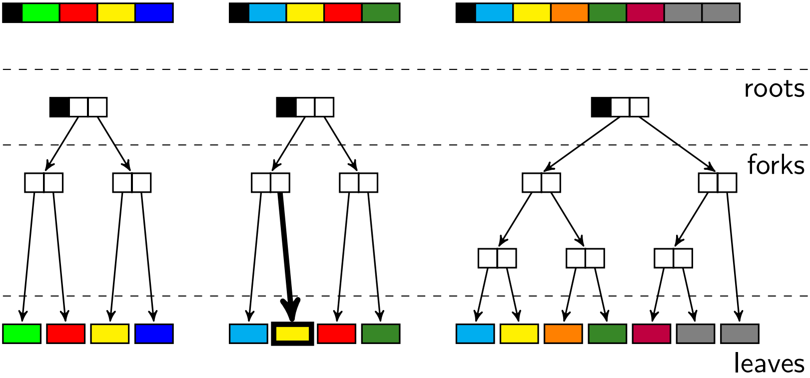
three types of tree nodes: roots – fork + associated data



Tree Compression

uses redundancy between states:

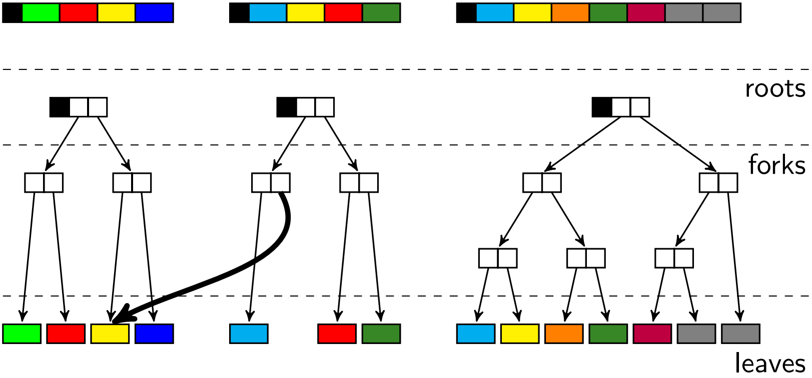
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

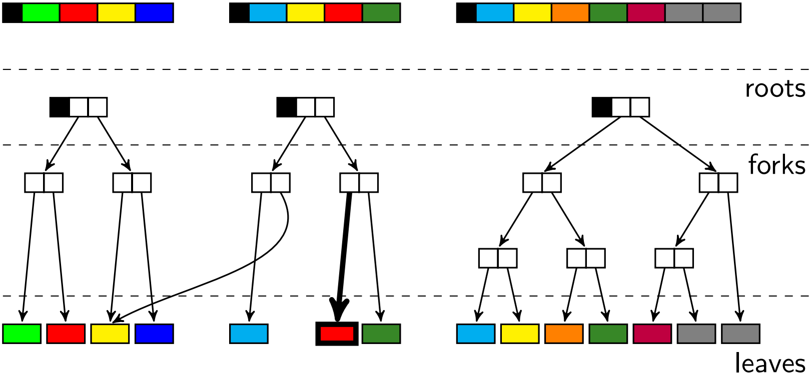
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

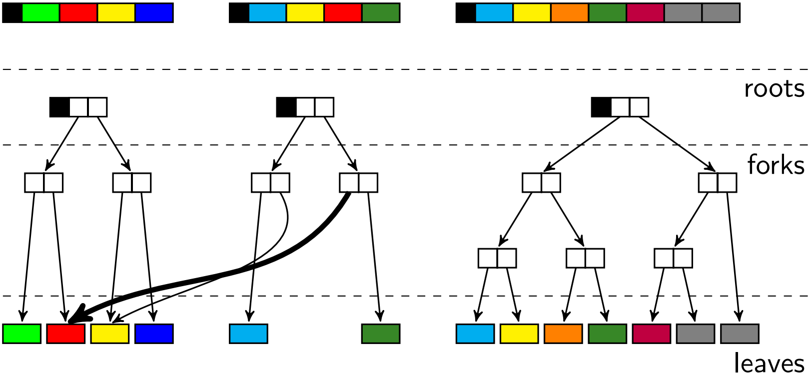
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

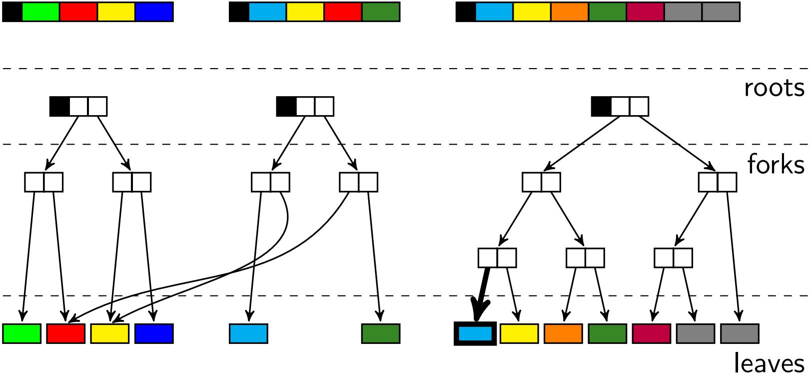
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

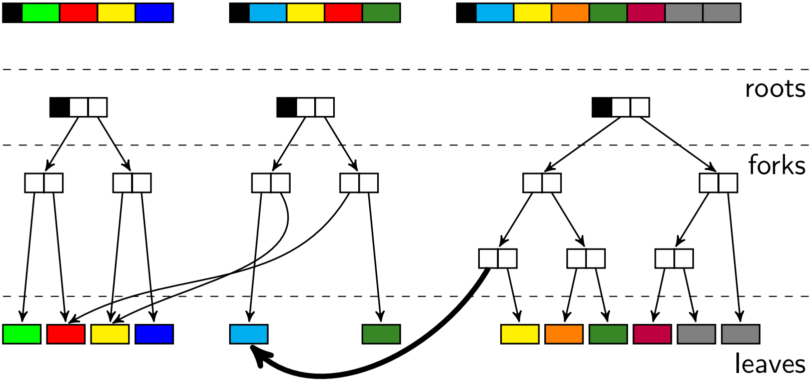
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

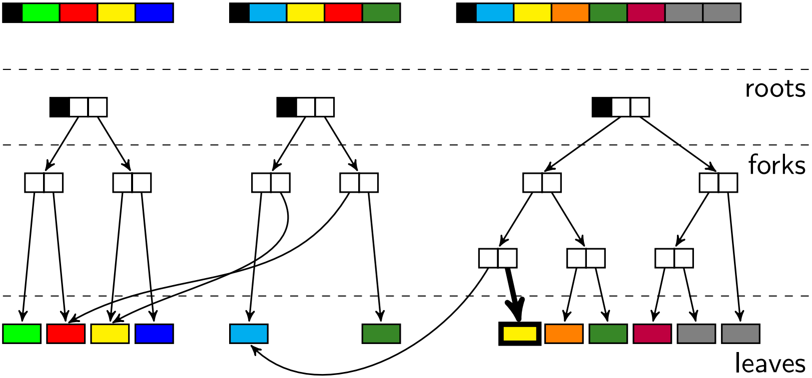
nodes are deduplicated using hash tables, one for each type



Tree Compression

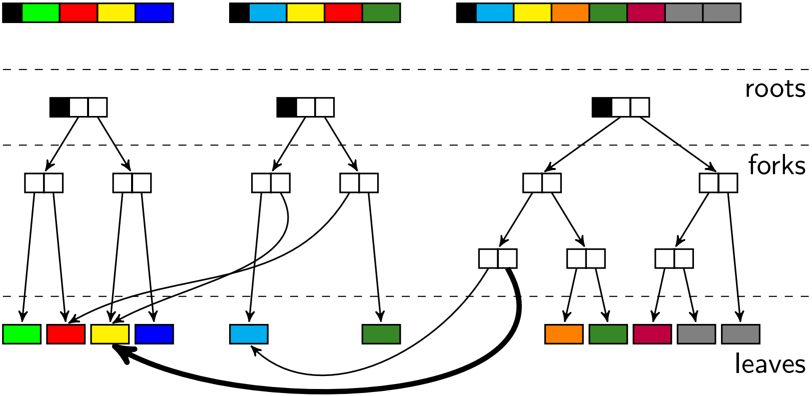
uses redundancy between states:

nodes are deduplicated using hash tables, one for each type



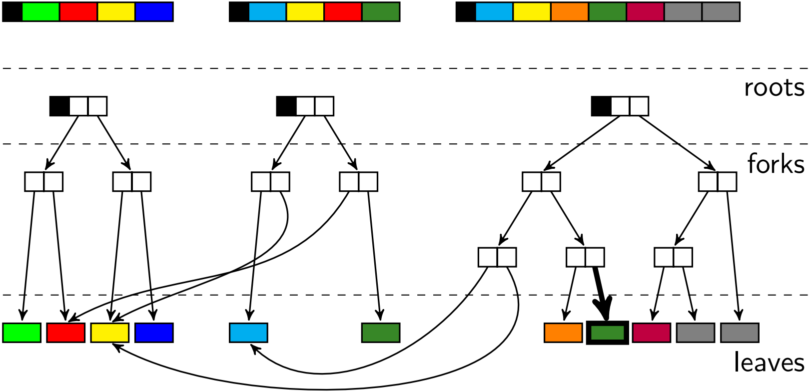
uses redundancy between states:

nodes are deduplicated using hash tables, one for each type



uses redundancy between states:

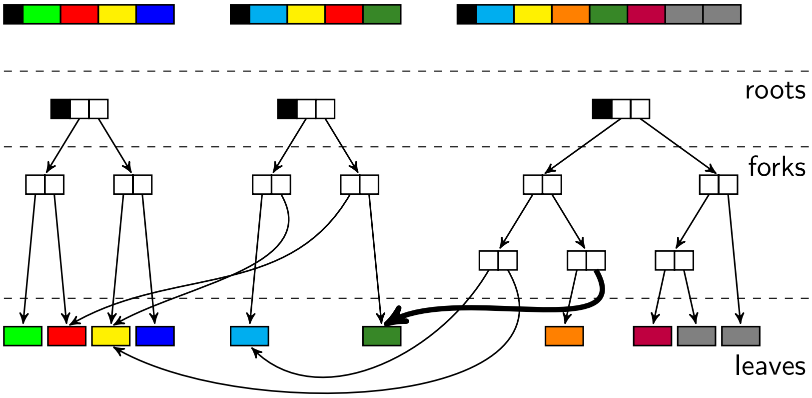
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

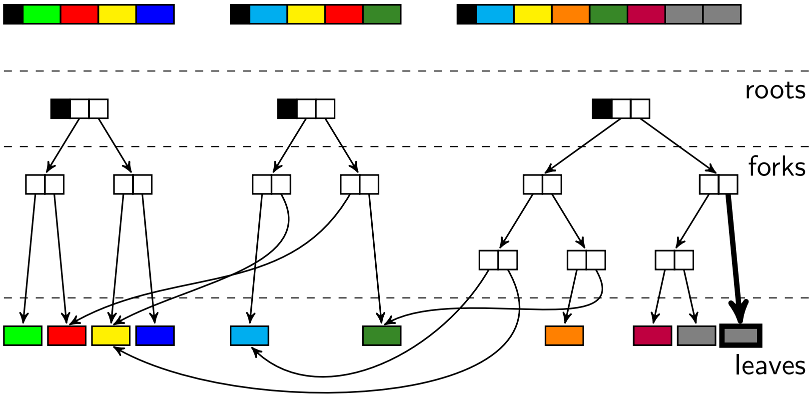
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

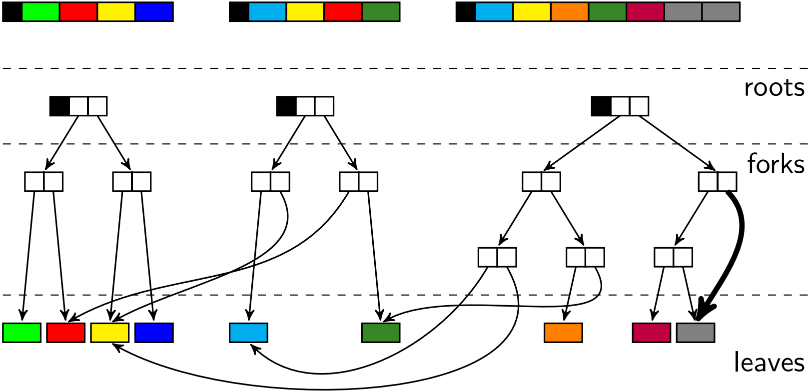
nodes are deduplicated using hash tables, one for each type



Tree Compression

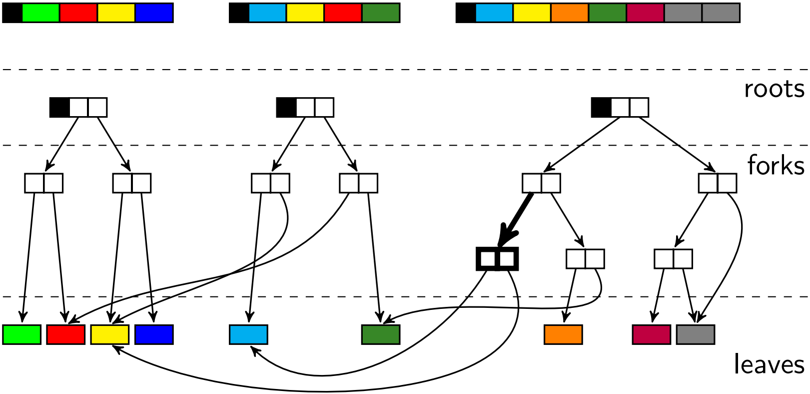
uses redundancy between states:

nodes are deduplicated using hash tables, one for each type



uses redundancy between states:

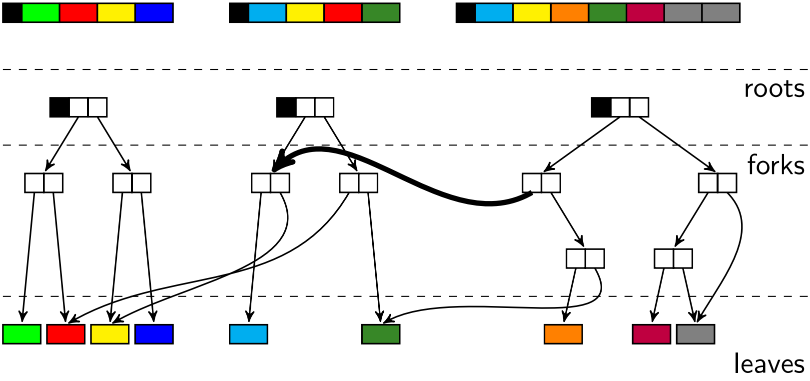
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

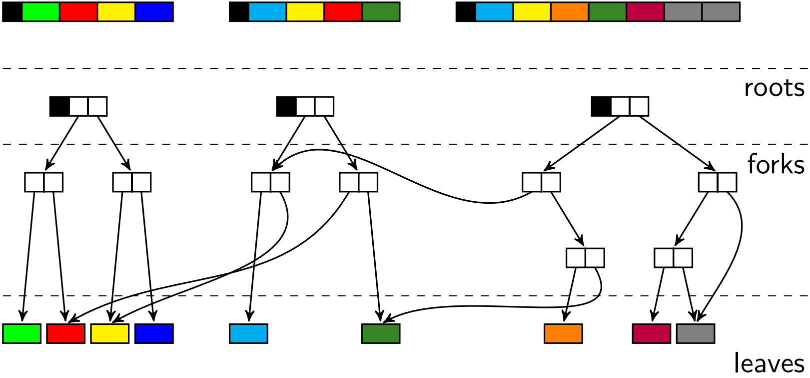
nodes are deduplicated using hash tables, one for each type



Tree Compression

uses redundancy between states:

nodes are deduplicated using hash tables, one for each type



Tree Compression

the design of a tree-compressed hash table

- a state is compressed when inserted
- roots are stored using the hash of the entire state in the root table
- the original state can be easily reconstructed by tree traversal

Tree Compression

the design of a tree-compressed hash table

- a state is compressed when inserted
- roots are stored using the hash of the entire state in the root table
- the original state can be easily reconstructed by tree traversal

- **if the underlying hash table is concurrent and resizeable so is the tree compressed table**

Tree Compression

the design of a tree-compressed hash table

- a state is compressed when inserted
- roots are stored using the hash of the entire state in the root table
- the original state can be easily reconstructed by tree traversal

- **if the underlying hash table is concurrent and resizeable so is the tree compressed table**

- **queue compressed (pointers to root nodes)**

Tree Compression

the design of a tree-compressed hash table

- a state is compressed when inserted
- roots are stored using the hash of the entire state in the root table
- the original state can be easily reconstructed by tree traversal

- **if the underlying hash table is concurrent and resizeable so is the tree compressed table**

- **queue compressed (pointers to root nodes)**

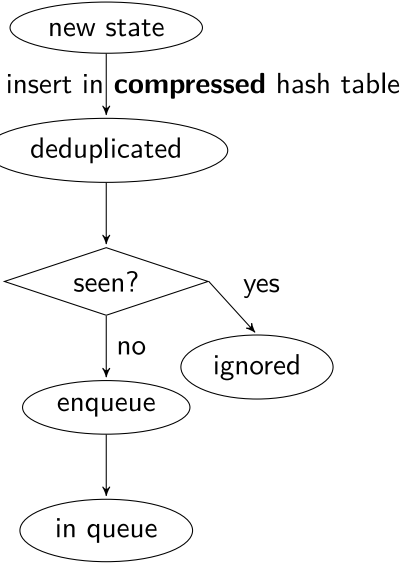
- the state space generator can direct the splitting of the state
  - need not be binary or balanced
  - but works well even without any modification to the generator

Tree Compression

the design of a tree-compressed hash table

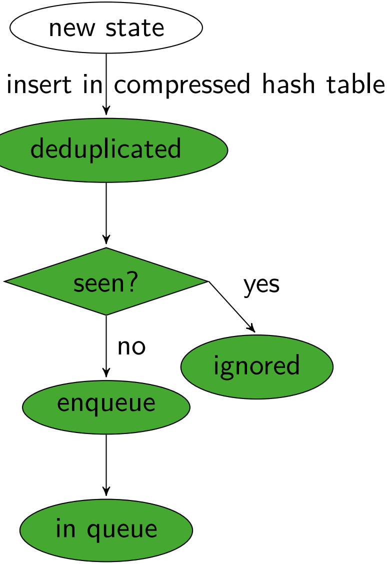
- a state is compressed when inserted
- roots are stored using the hash of the entire state in the root table
- the original state can be easily reconstructed by tree traversal
- **if the underlying hash table is concurrent and resizeable so is the tree compressed table**
- **queue compressed (pointers to root nodes)**
- the state space generator can direct the splitting of the state
  - need not be binary or balanced
  - but works well even without any modification to the generator
- works better for larger state spaces

State Processing with Compression



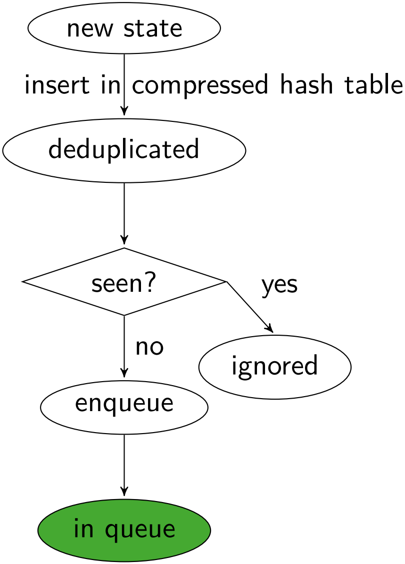
- state is compressed on insertion into hash table

State Processing with Compression



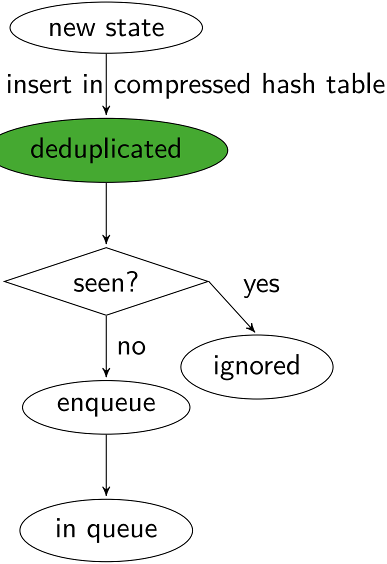
- state is compressed on insertion into hash table
- most of the time state is compressed

State Processing with Compression



- state is compressed on insertion into hash table
- most of the time state is compressed
- **queue is compressed**

State Processing with Compression



- state is compressed on insertion into hash table
- most of the time state is compressed
- queue is compressed
- original state is always deallocated when compressed

Memory Allocation: Scheme

- DIVINE allocates many memory blocks
- blocks which store compressed nodes only came in limited number of sizes
- blocks generated by the state space generator are short-lived and came in many different sizes
- freed blocks should be reused

Memory Allocation: Scheme

- DIVINE allocates many memory blocks
- blocks which store compressed nodes only came in limited number of sizes
- blocks generated by the state space generator are short-lived and came in many different sizes
- freed blocks should be reused

- the generator, the compression scheme, and the allocator all need to know the block size

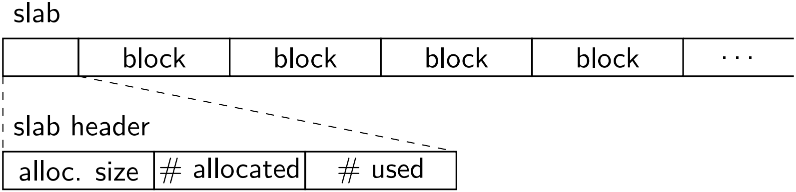
Memory Allocation: Scheme

- DIVINE allocates many memory blocks
- blocks which store compressed nodes only came in limited number of sizes
- blocks generated by the state space generator are short-lived and came in many different sizes
- freed blocks should be reused

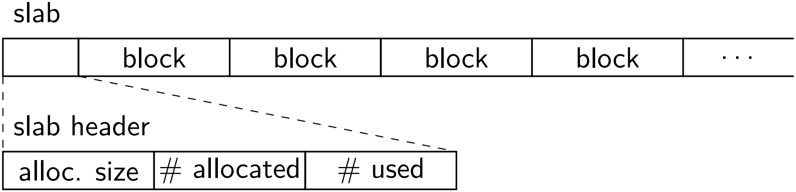
- the generator, the compression scheme, and the allocator all need to know the block size
  - only the allocator can store it compactly

Memory Allocation: DIVINE's Allocator

- allocates memory in large slabs
  - used for the allocation of same-sized memory blocks
  - remembers the size in each slab
- memory is addressed indirectly (slab address + offset)
- uses free-lists for memory reuse



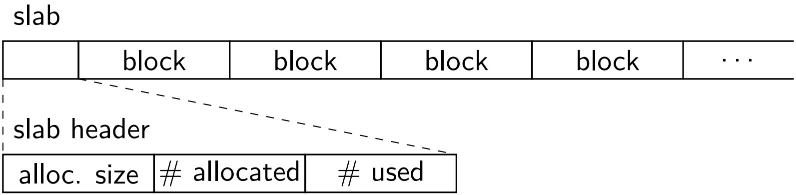
- allocates memory in large slabs
  - used for the allocation of same-sized memory blocks
  - remembers the size in each slab
- memory is addressed indirectly (slab address + offset)
- uses free-lists for memory reuse



- works well for tree-compressed nodes

Memory Allocation: DIVINE's Allocator

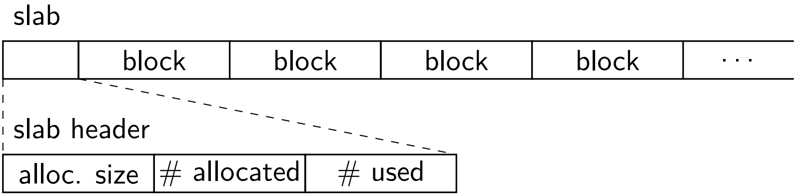
- allocates memory in large slabs
  - used for the allocation of same-sized memory blocks
  - remembers the size in each slab
- memory is addressed indirectly (slab address + offset)
- uses free-lists for memory reuse



- works well for tree-compressed nodes
- short-lived blocks need optimization

Memory Allocation: DIVINE's Allocator

- allocates memory in large slabs
  - used for the allocation of same-sized memory blocks
  - remembers the size in each slab
- memory is addressed indirectly (slab address + offset)
- uses free-lists for memory reuse

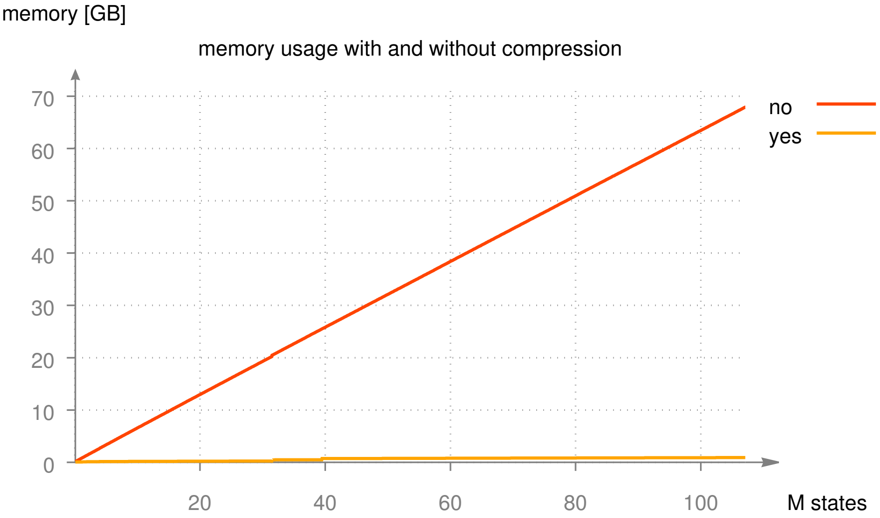


- works well for tree-compressed nodes
- short-lived blocks need optimization
  - allocated in a special slab for ephemeral memory

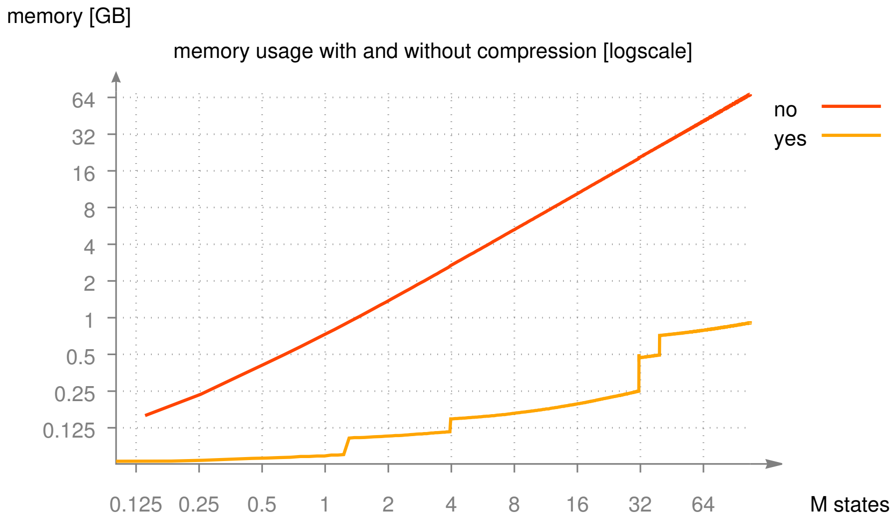
Memory Allocation: Pointer Size

- 32 bit pointers – compact but limiting
- 64 bit pointers – in fact 48 bits used on x86-64
  - the rest is reserver and should not be used
- additional storage in pointer would be useful
- our indirect pointers: 39 bit pointer + 25 bit tag
- the tag can be used by the tree compression scheme (to distinguish forks/leaves) and by the hash table (to help in collision resolution)

Results: C and C++ models



Results: C and C++ models



Results: C and C++ models

Name	# of states	memory usage (GB) compression no	yes	**ratio**
pt_rwlock	10.7 M	68	0.88	**77**×
pt_barrier	128.5 M	> 825	5.48	**151**×
collision	3.0 M	48	0.64	**74**×
elevator2	33.0 M	> 343	2.50	**137**×
lead-uni_basic	19.2 M	232	0.81	**288**×
lead-uni_peterson	12.2 M	146	0.64	**230**×
hashset-2-4-2	6.7 M	133	1.20	**111**×
hashset-3-1	626.9 M	> 15 110	27.51	**549**×

Conclusion, Future Work

Conclusion

- enables verification of real-world code
- large memory savings (74-550$\times$)
- on top of saving from $\tau+$ reduction (50-1000$\times$)
- decent performance (no more that 2$\times$ slower)
- full parallel verification supported with compression code

Conclusion, Future Work

Conclusion

- enables verification of real-world code
- large memory savings (74-550$\times$)
- on top of saving from $\tau+$ reduction (50-1000$\times$)
- decent performance (no more that 2$\times$ slower)
- full parallel verification supported with compression code

Future work

- more efficient distributed compression
- performance optimizations

Conclusion, Future Work

Conclusion

- enables verification of real-world code
- large memory savings (74-550$\times$)
- on top of saving from $\tau+$ reduction (50-1000$\times$)
- decent performance (no more that 2$\times$ slower)
- full parallel verification supported with compression code

Future work

- more efficient distributed compression
- performance optimizations

`http://divine.fi.muni.cz`

Thank you!