Towards Improving the Resource Usage of SAT Solvers

Analyzing and Improving the Resource Usage of a State of the Art SAT Solver

Norbert Manthey[1]

10.07.2010

[1]supported by the EMCL (European Master's Program in Computational Logic)

Outline

Motivation - Why improve SAT Solver?

SAT Solvers can solve several problems

(Bounded Model Checking, Planning, Software Verification, ...)

How can sequential SAT solving be potentially improved?

- No knowledge about resource utilization
- No obvious metric to choose the best algorithm

Optimized version: in average *only 40%* of original runtime

SAT Solving

Given: Conjunction of clauses (special cases: Unit, Binary)

Task: Find satisfying assignment for variables if possible.

Industrial problems: millions of variables and clauses (SAT Comp. 2009).

Used Solver: riss, 4400 lines C++, 64 bit

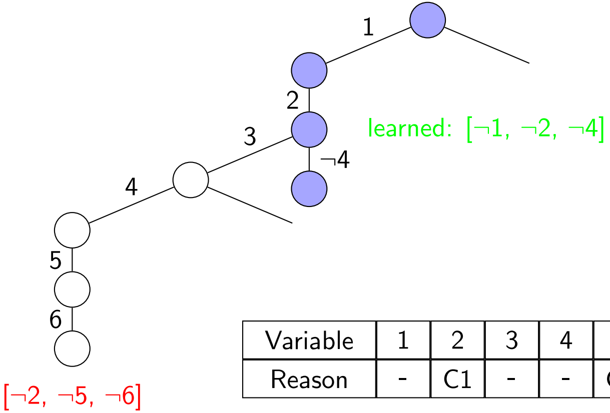
successor version qualified for SAT Race 2010

Used Techniques (only relevant mentioned):

- Two-Watched-Literal Unit Propagation
- Special treatment of binary clauses
- Conflict Analysis, Learning and Backjumping

Finding an Satisfying (Partial) Assignment

Using binary search tree. Question: Which clause to check next?



Variable	1	2	3	4	5	6
Reason	-	C1	-	-	C2	C3

F=⟨ [¬1, 2], [¬4, 5], [¬1, ¬4, 6], [¬2, ¬5, ¬6], [1, 3] ⟩

Modern Memory Resources

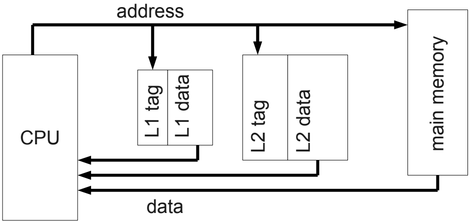
Facts:

- SAT Solving involves lots of memory (avg. 220 MB)
- No easy memory access pattern
- Aim: improve speed of memory accesses
- Utilize CPUs memory units better

Memory Hierarchy and Units:

- Main Memory
- Caches
- Prefetching Unit
- Translation Lookaside Buffers

Accessing Data in the Memory Hierarchy



Level	Size	Latency (in cycles)
Main Memory	2 GB	240
L2 Cache	1 MB	14
L1 Cache	64 KB + 64 KB	3

organized in lines (64 bytes)

Prefetch Memory into Cache

Prefetching Unit:

- Fetches data into the cache
- Works in parallel to algorithm execution
- Usually controlled by hardware (simple patterns)
- Can be controlled by software instructions

Pro:

- Reduces time to wait for main memory
- Does not introduce additional latency

Contra:

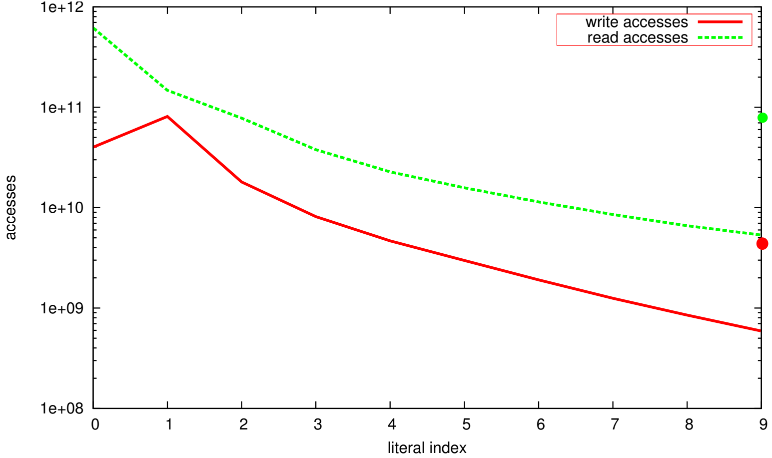
- Prefetching unnecessary data may evict important data

Resource Consumption

	Cycles	Stall Cycles	L2 Misses	L2 Accesses
Program	100.0%	100.0%	100.0%	100.0%
Other Components	2.01%	1.80%	3.22%	3.16%
Conflict Analysis	5.74%	5.42%	6.27%	7.27%
Propagation	91.65%	92.62%	90.08%	88.94%
Propagate binary	5.71%	5.55%	7.95%	5.64%
Propagate long	83.86%	85.30%	78.17%	79.78%
Literal read access	45.80%	54.49 %	24.07%	12.57%
Maintain Watch List	24.26%	18.59%	2.19%	36.64%

Wait Rate: 82%, L2 Miss Rate: 40%

Literal Access Distribution

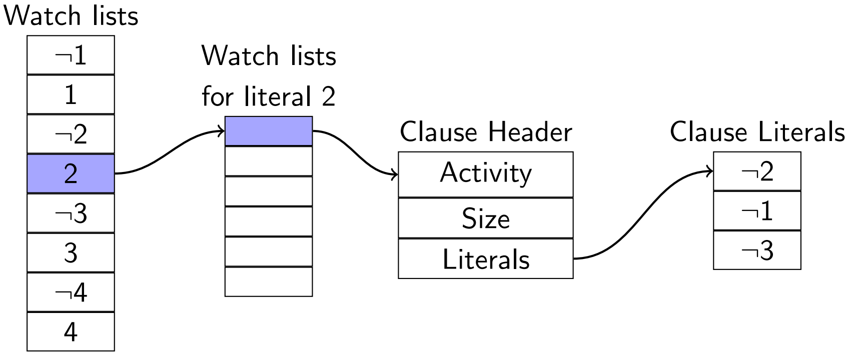


Read Access: 60% on literal 0, 15% on literal 1, decreasing

Write Access: 25% on literal 0, 50% on literal 1, decreasing

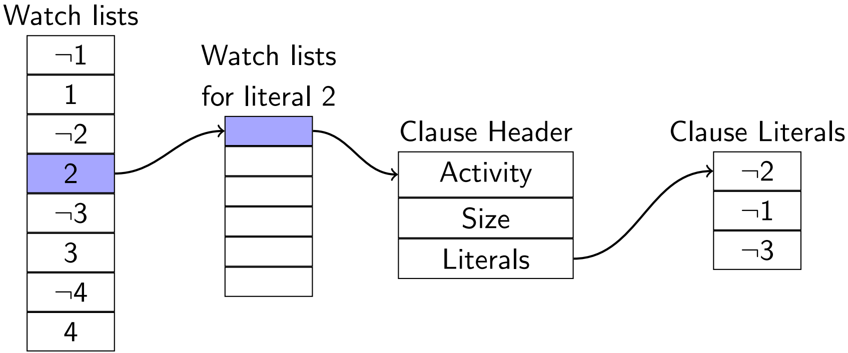
Ratio: Write Accesses is sixth part of Read Accesses

Apply Knowledge to Implementation



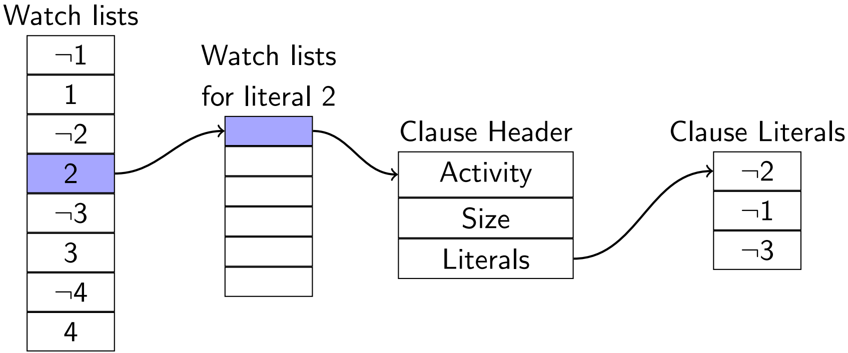
- *Prefetching*: Prefetch all clauses of watched list
- *Flattened Clause*: Combine Clause Header and Clause Literals
- *Cache Clause*: Store a few literals in Clause Header

Apply Knowledge to Implementation



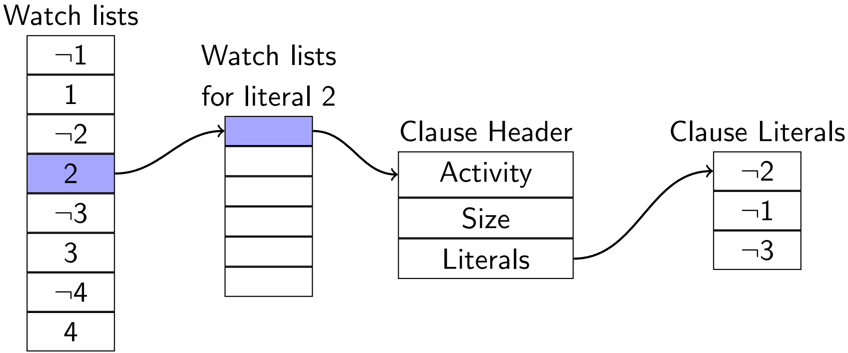
- *Prefetching*: Prefetch all clauses of watched list
- *Flattened Clause*: Combine Clause Header and Clause Literals
- *Cache Clause*: Store a few literals in Clause Header

Apply Knowledge to Implementation



- *Prefetching*: Prefetch all clauses of watched list
- *Flattened Clause*: Combine Clause Header and Clause Literals
- *Cache Clause*: Store a few literals in Clause Header

Apply Knowledge to Implementation



- *Prefetching*: Prefetch all clauses of watched list
- *Flattened Clause*: Combine Clause Header and Clause Literals
- *Cache Clause*: Store a few literals in Clause Header

Further Improvements

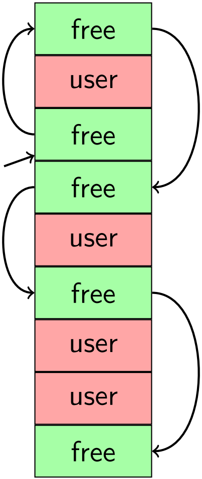
Reduce memory overhead

- Avoid System Allocator space overhead, use Slab Allocator
- Compress Boolean array and Assignment
- Compress literals in clause

Reduce memory accesses

- Remove elements lazily from vector (*Lazy Removal*)
- Reuse vectors instead of recreation (*Reuse Vector*)

Slab Allocator



Properties:

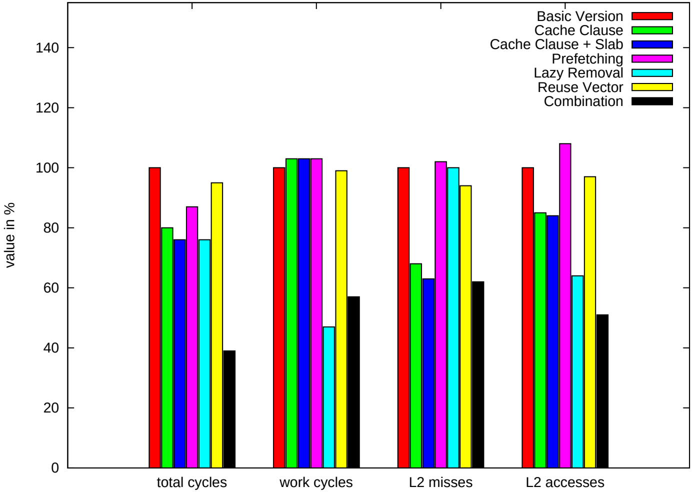
- Allocates big memory blocks
- Separate them into slabs of fixed slab size
- No overhead between slabs
- Keeps track of free slabs (linked list)

Used slabs: User knows address, uses storage

Free slabs: Allocator uses storage for linked list

Suitable to store two Clause Headers on a single Cache Line

Overview of Improvements and Combinations



Conclusion and Future Work

All presented improvements do not change search (micro optimization).

Rules to follow:

1. Increase access locality
2. Reduce number of memory accesses (cache line loads)
3. Use prefetching for difficult access pattern
4. use 2 MB pages (additional 10% improvement)

Future Work:

- Analyze costs of Branch Miss-Prediction, effects on Cache Misses
- Analyze effects of improvements on parallel solvers

Micro optimized solver needs 40% on average. Implementation is important.

Slab Allocator, Prefetching are not used in another solver.

Thanks

Co-author: Ari Saptawijaya

SAT Solving Group:

Christoph Baldow, Friedrich Gräter,

Max Seeleman, Steffen Hölldobler

Operating System Group:

Hermann Härtig, Julian Stecklina