Towards Improving the Resource Usage of SAT Solvers

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

Norbert Manthey¹

10.07.2010

¹supported by the EMCL (European Master’s Program in Computational Logic)
Outline

1 Introduction
2 Modern Memory Resources
   - Caches
   - Prefetching Unit
3 Analysis and Improvements
   - Major Improvements
   - Further Improvements
   - Overall Results
4 Conclusion
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?

\[ F = \langle [-1, 2], [-4, 5], [-1, -4, 6], [-2, -5, -6], [1, 3] \rangle \]

<table>
<thead>
<tr>
<th>Variable</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>-</td>
<td>C1</td>
<td>-</td>
<td>-</td>
<td>C2</td>
<td>C3</td>
</tr>
</tbody>
</table>

learned: \([-1, -2, -4]\)
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

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>Latency (in cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main Memory</td>
<td>2 GB</td>
<td>240</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1 MB</td>
<td>14</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>64 KB + 64 KB</td>
<td>3</td>
</tr>
</tbody>
</table>

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

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

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

Watch lists

-1
1
-2
2
-3
3
-4
4

Watch lists for literal 2

Clause Header

<table>
<thead>
<tr>
<th>Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
</tr>
<tr>
<td>Literals</td>
</tr>
</tbody>
</table>

Clause Literals

-2
-1
-3

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

Flattened Clause: Combine Clause Header and Clause Literals

Cache Clause: Store a few literals in Clause Header
Watch lists for literal 2

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

### Flattened Clause
Combine Clause Header and Clause Literals

### Cache Clause
Store a few literals in Clause Header

---

<table>
<thead>
<tr>
<th>Watch lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \neg 1 )</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>( \neg 2 )</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>( \neg 3 )</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>( \neg 4 )</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

Watch lists for literal 2

- Clause Header
  - Activity
  - Size
  - Literals

- Clause Literals
  - \( \neg 2 \)
  - \( \neg 1 \)
  - \( \neg 3 \)
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

![Bar chart showing improvements and combinations for total cycles, work cycles, L2 misses, and L2 accesses.](chart.png)

- Basic Version
- Cache Clause
- Cache Clause + Slab
- Prefetching
- Lazy Removal
- Reuse Vector
- Combination

Norbert Manthey
Towards Improving the Resource Usage of SAT Solvers 10.07.2010 14 / 16
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