Lingeling Essentials

Design and Implementation Aspects

Armin Biere
Johannes Kepler University
Linz, Austria

POS 2014

5th Workshop on Pragmatics of SAT 2014

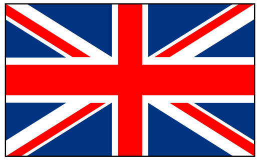
SAT 2014 / FLoC 2014
Vienna Summer of Logic

Vienna, Austria

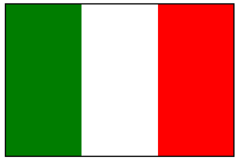
Sunday, 13 July, 2014

Lingeling successor of PrecoSAT (Inprocessing)

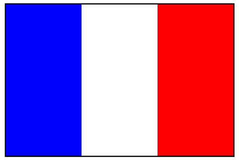
lightweight (compact), beautiful written in C



Butterfly



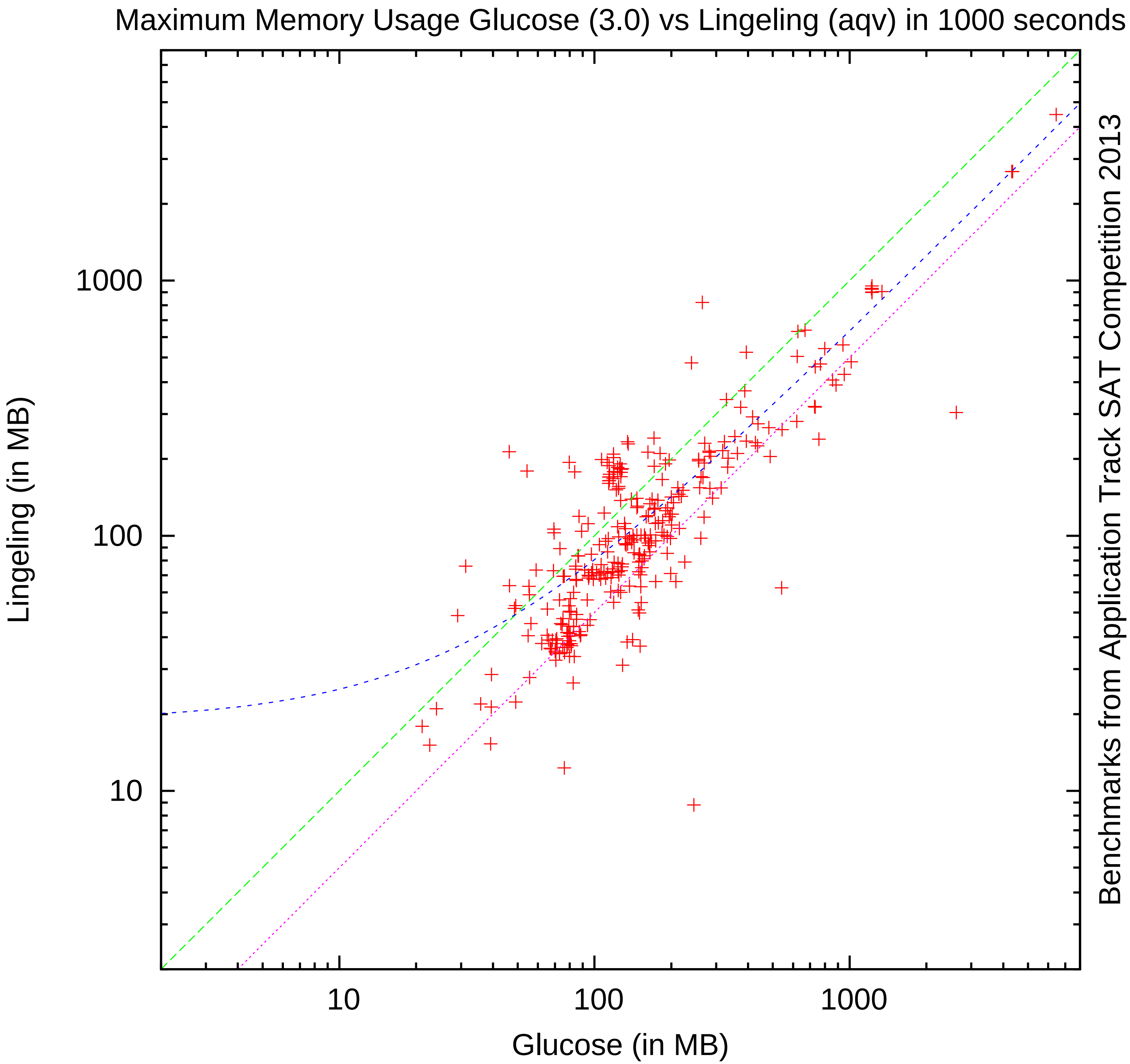
Farfalla



Papillon

Schmetterling

my 3 year old daughter used *Lingeling* instead of *Schmetterling*

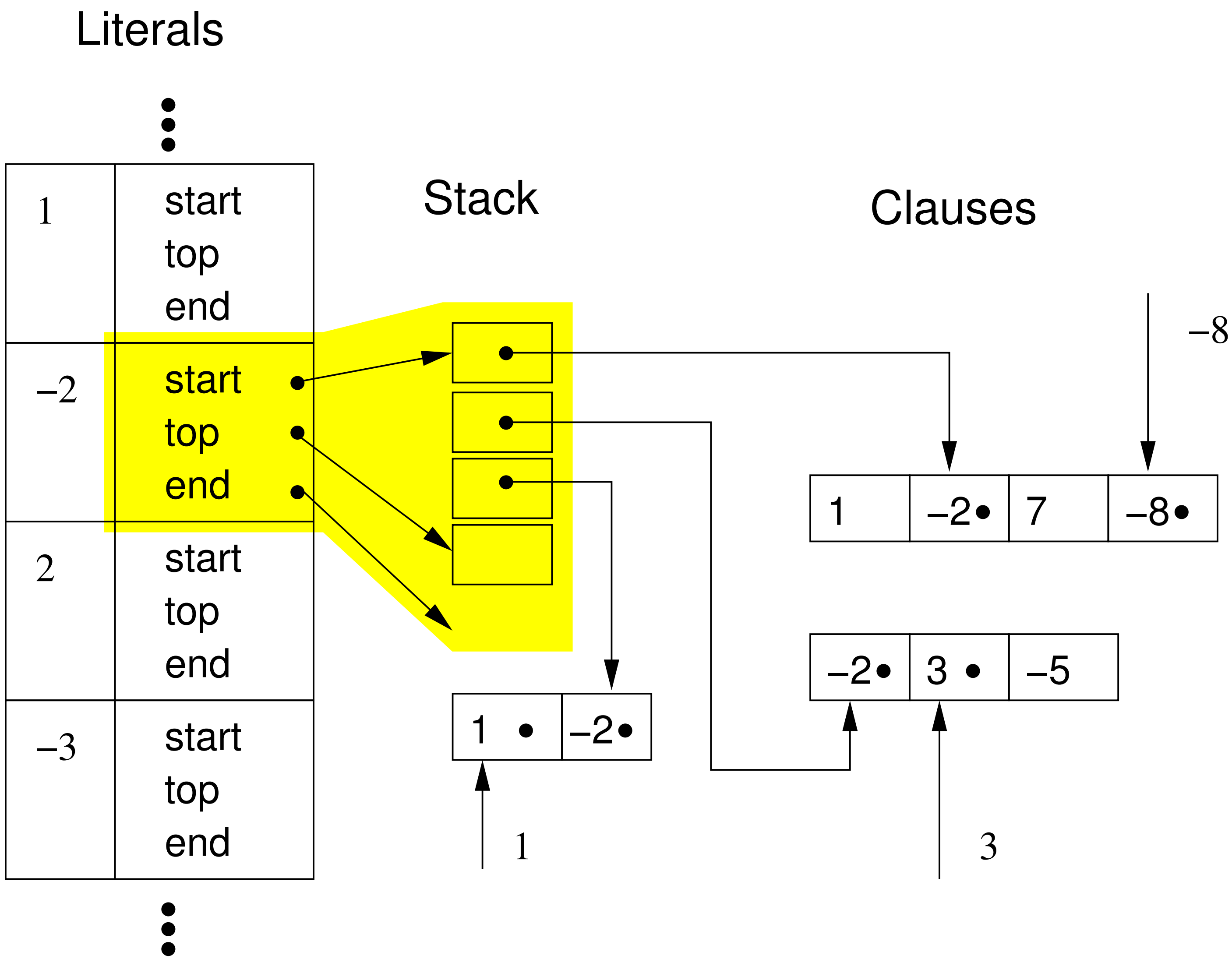


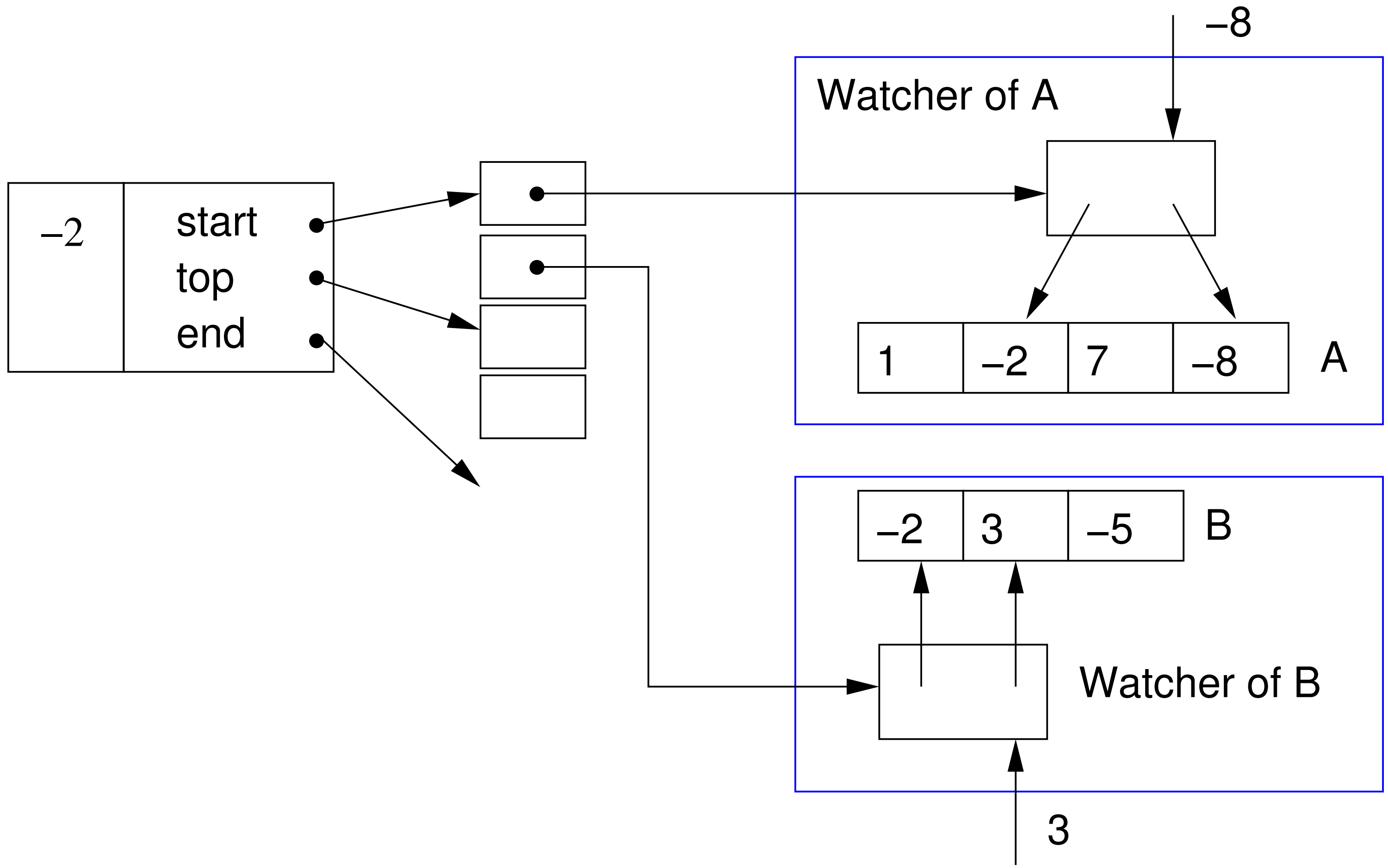
Maximum Memory Usage Glucose (3.0) vs Lingeling (aqv) in 1000 seconds
Lingeling (in MB)
1000
100
10
10
100
1000
Glucose (in MB)
Benchmarks from Application Track SAT Competition 2013

Compact Data Structures for CDCL

- focus on conflict-driven clause learning (CDCL)
  - similar arguments apply to look-ahead or local search solvers
  - preprocessing / inprocessing have to be considered as well
- memory usage dominated by clause data base
  - memory layout of individual clauses
  - occurrence lists of references to (watched) clauses
- cache friendliness
  - keep data compact (maximize what fits in a cache line)
  - minimize pointer dereferences (mems)
  - low-level parallelization not considered here
- watching clauses (sparse mode) versus full occurrence lists (dense mode)
- special treatment of short clauses: binary and ternary

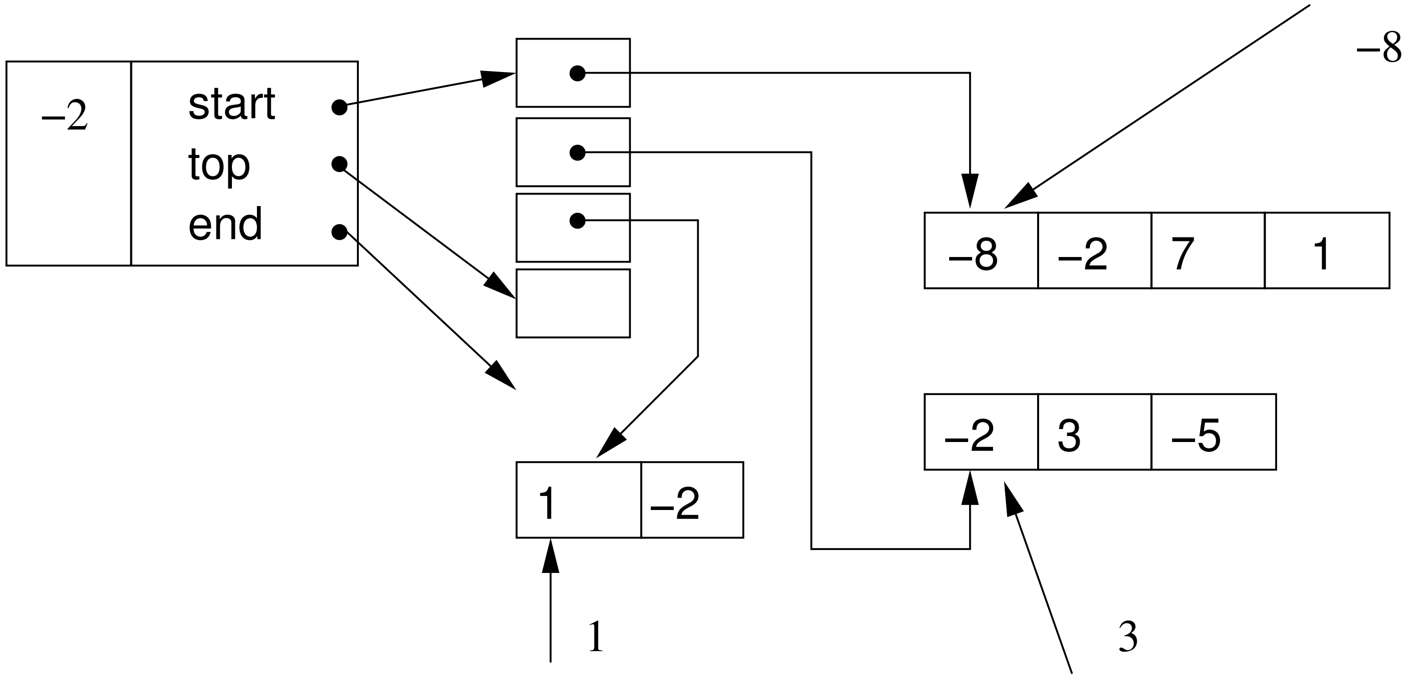
ZChaff Occurrence Stacks



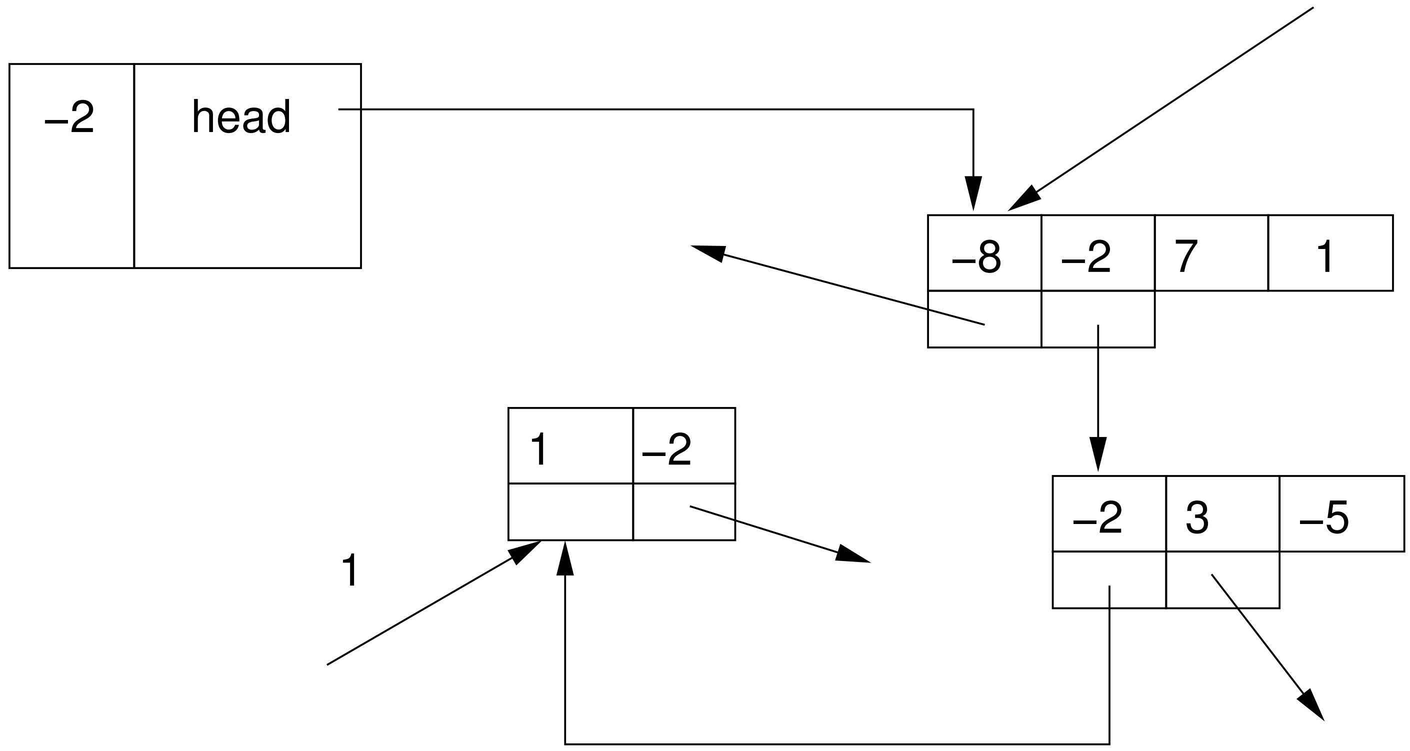


−8
Watcher of A
−2
start
top
end
1
−2
7
−8
A
−2
3
−5
B
Watcher of B
3

CompSAT / MiniSAT Occurrence Stacks



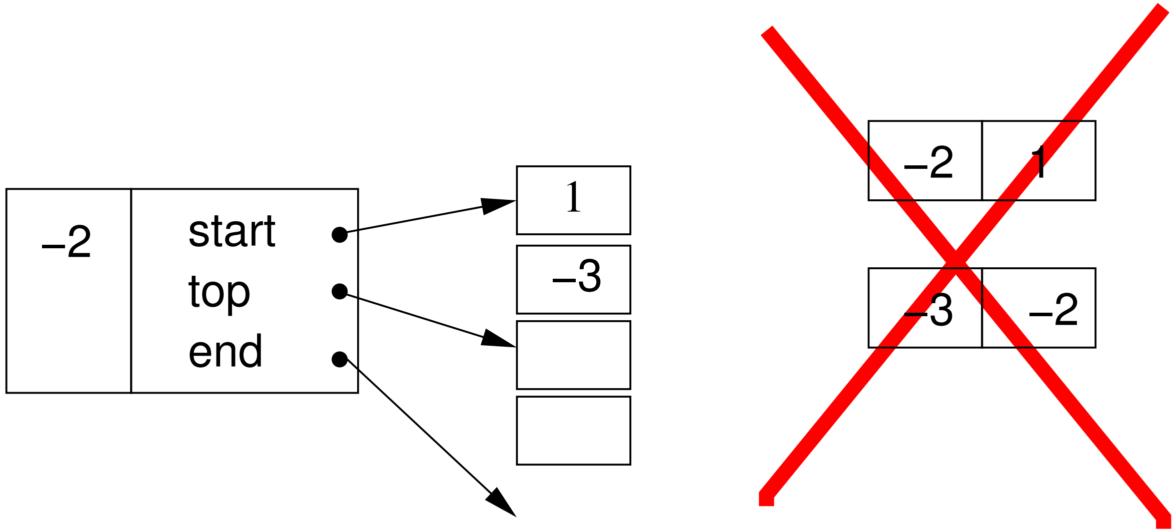
invariant: first two literals are watched



invariant: first two literals are watched

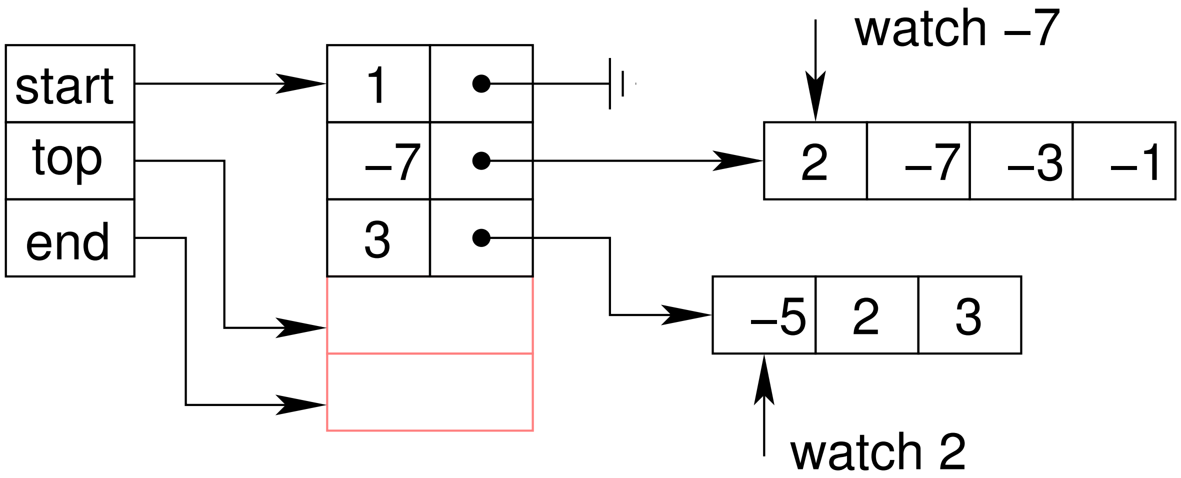
Occurrence Stacks for Binary Clauses

Additional Binary Clause Watcher Stack



Blocking Literals

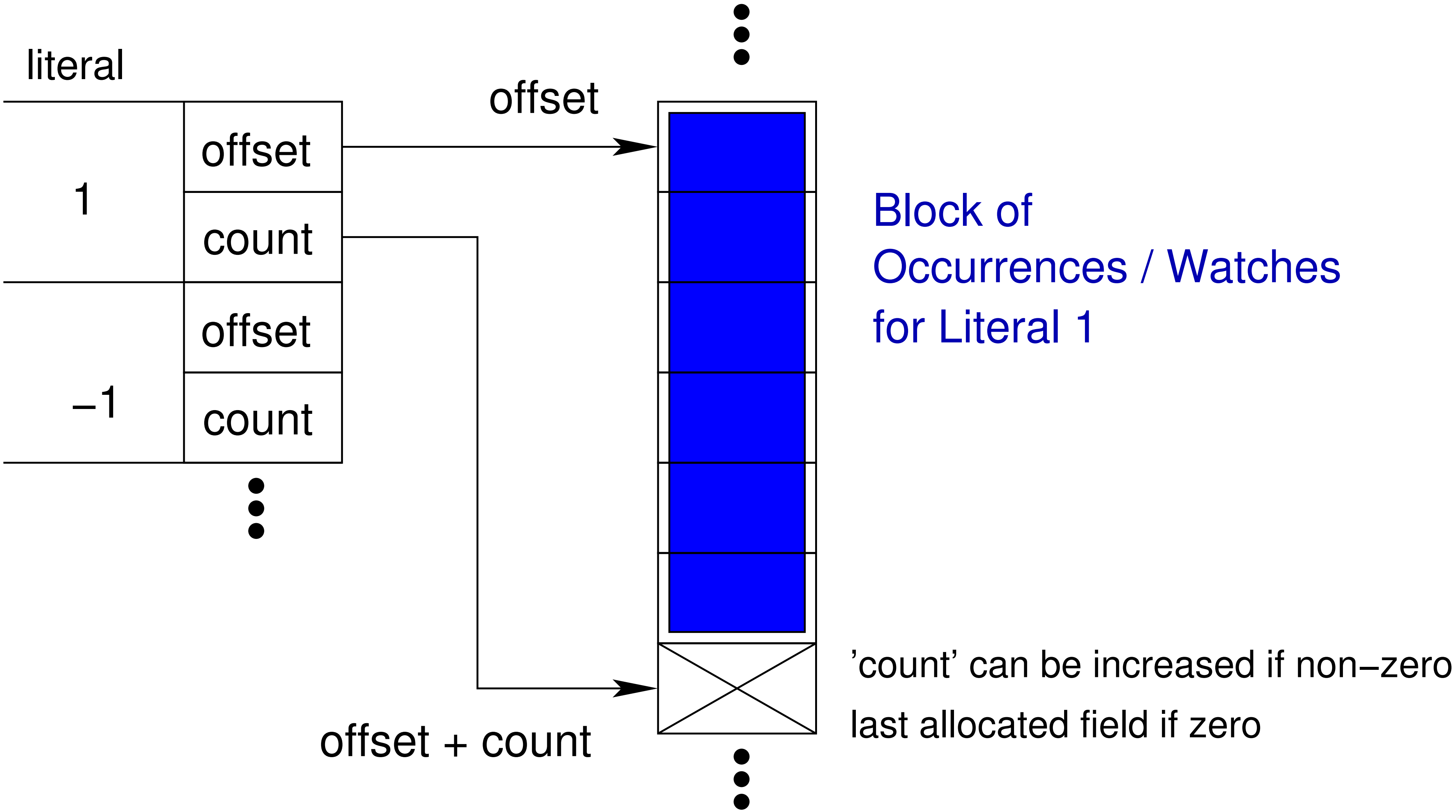
ChuHarwoodStuckey'09



- observation: often the *other* watched literal satisfies the clause
  - so cache these literals in watch list to avoid pointer dereference
- for binary clause no need to store clause at all
  - never has to access the actual clause data
  - needs special treatment of binary clauses during conflict analysis
  - reasons are either references to clauses or “other” literals of binary clauses
- can easily be adjusted for ternary clauses
  - with full occurrence lists (all three literals are watched)
  - a ternary reason consists of the “other two” literals

Lingeling Occurrence Lists

two 32–bit integer stacks



Lingeling Occurrence Lists

- assumes number of watches much smaller than 2^{32}
  - actually closer to 2 billion, but still very reasonable in practice
  - the `count` field is needed for fast “pushing of watches”
- 8 bytes for offset/count entry per literal
  - plus 4 bytes for sentinel on the actual watches stack
  - MiniSAT / Glucose / STL Stack need 3 pointers (24 bytes on 64-bit machine)
- contiguous occurrences / watches stack needs explicit memory management
  - without contiguous memory need pointer instead of offset (so 64 bit)
  - if occurrence / watch pushed and (blue) block full for this literal reallocate
  - maintain free lists of free blocks
  - might need to reallocate (with `realloc`) whole stack of blocks
    - which in turn might move addresses of the (blue) blocks
    - so pushing watches while iterating (blue) blocks dangerous
  - periodical defragmentation of blocks to keep overhead small

Literal Stacks

- actual clause data stored on literal stacks (only clauses with at least 4 literals)
  - first two literals are watched
  - integer literals separated by zero sentinels (think DIMACS format)
  - learned clauses have an additional 32-bit activity counter (before the actual literals)
- separate stacks for redundant (original) clauses and irredundant (learned) clauses
  - we cluster learned clauses with similar glucose level (LBD) into 16 clusters
  - each cluster corresponds to one “scaled glue” and has one literal stack
- references to clauses are actually offsets into these stacks
  - pushing clauses while iterating through literals is dangerous
  - restricts number of literals in each cluster to 2^{32}

```
irr     -1 2 3 4 0 5 1 6 -4 9 0 ...
red[0]  47536 6 -3 4 7 8 2 0 4789 -6 -3 7 8 2 5 0 ...
....
red[14] ...
```

- `MAXGLUE = 15` clauses are actually discarded after backtracking

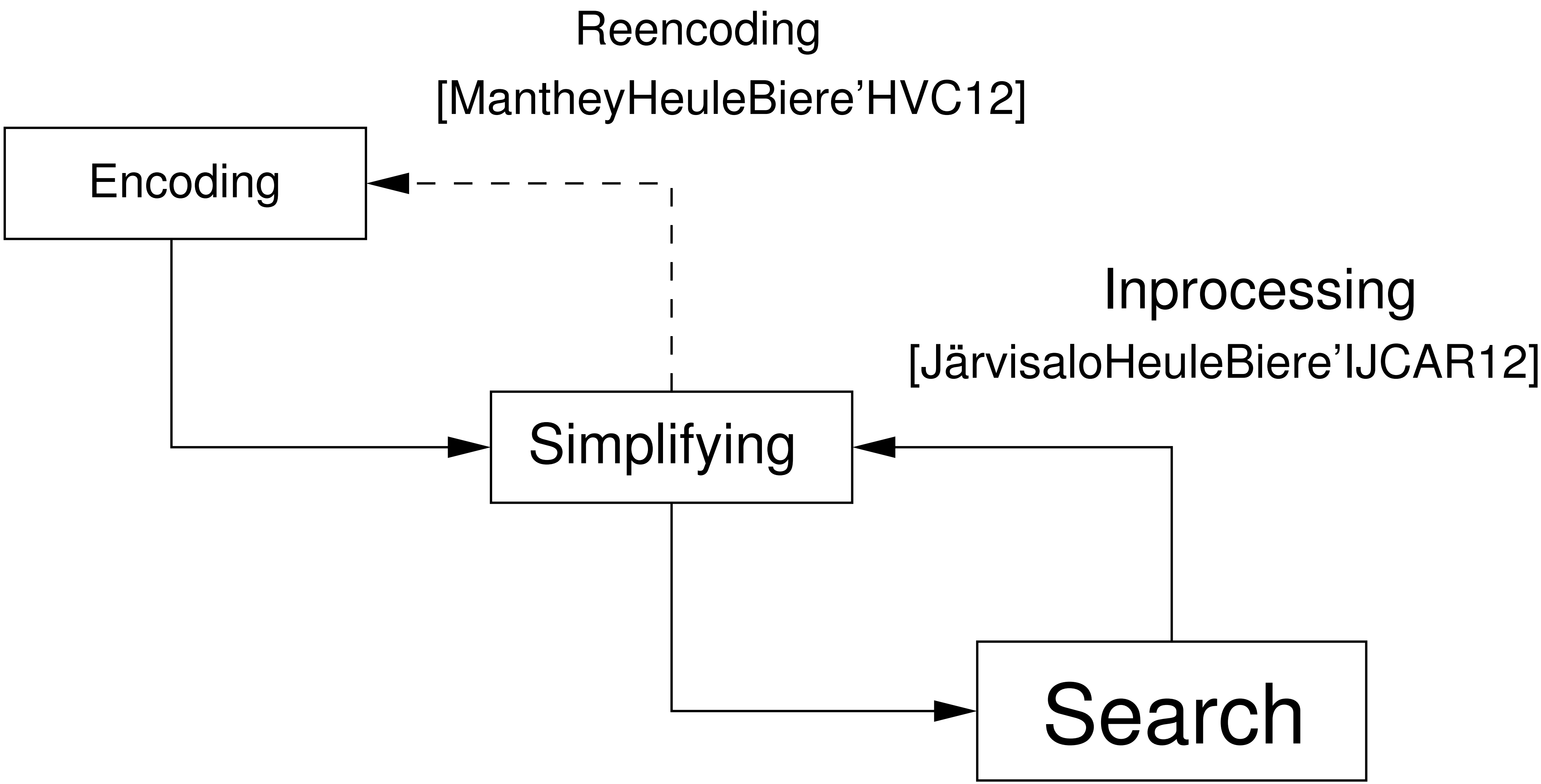
- entries in occurrence list are classified as
  - *binary*, *ternary*, *large* watch, large *occurrence* (constraint types)
  - redundant or irredundant clause (redundancy)
- constraint types are used for classifying reasons too
  - need two additional types: *unit* clause, *decision*
  - altogether 3 bits are used to encode the constraint type
- one bit is used to encoded redundancy
  - binary and ternary clauses are only stored in occurrence lists
  - during preprocessing it is essential to know their redundancy
- remaining 28 = 32 - 4 bits of first integer used to encode blocking literal / occurrence
  - restriction on a maximum of $2^{27} = 134$ million variables
  - and the same number of actual literals in irredundant clauses (including sentinels)
- ternary clauses have an additional blocking literal (wasting four bits)
- large watched clauses have and additional offset into literal stack
  - for irredundant clauses the glucose level is stored in least significant four bits

- binary clauses
  - `3.0.2` (hexadecimal `0000 0032`)
    reference to a irredundant binary clause with other literal `3`
  - `-2.1.2` (hexadecimal `ffff ffea`)
    reference to a redundant binary clause with other literal `-2`
- ternary clauses
  - `7.0.3 -1` (hexadecimal `0000 0073 ffff ffff`)
    reference to a irredundant ternary clause with other literals `7` and `-1`
- large watched clauses
  - `5.0.4 9` (hexadecimal `0000 0054 0000 0009`)
    reference to large watched irredundant clause, blocking literal `5`, offset `9`
  - `6.1.4 12.8` (hexadecimal `0000 006b 0000 00c8`)
    reference to large watched redundant clause, blocking literal `6`, glue `12`, offset `8`
- large occurrence
  - `17.0.1` (hexadecimal `0000 0111`)
    reference to large clause with offset `17` in irredundant literal stack

Inprocessing: Interleaving Preprocessing and Search

PrecoSAT [Biere'09], Lingeling [Biere'10], also in CryptoMiniSAT (Mate Soos)

- preprocessing can be extremely beneficial
  - most SAT competition solvers use bounded variable elimination (BVE) [EénBiere SAT'05]
  - equivalence / XOR reasoning
  - various clause elimination procedures
  - probing / failed literal preprocessing / hyper binary resolution
  - however, even though polynomial, can not be run until completion
- simple idea to benefit from full preprocessing without penalty
  - "preempt" preprocessors after some time
  - resume preprocessing between restarts
  - limit preprocessing time in relation to search time

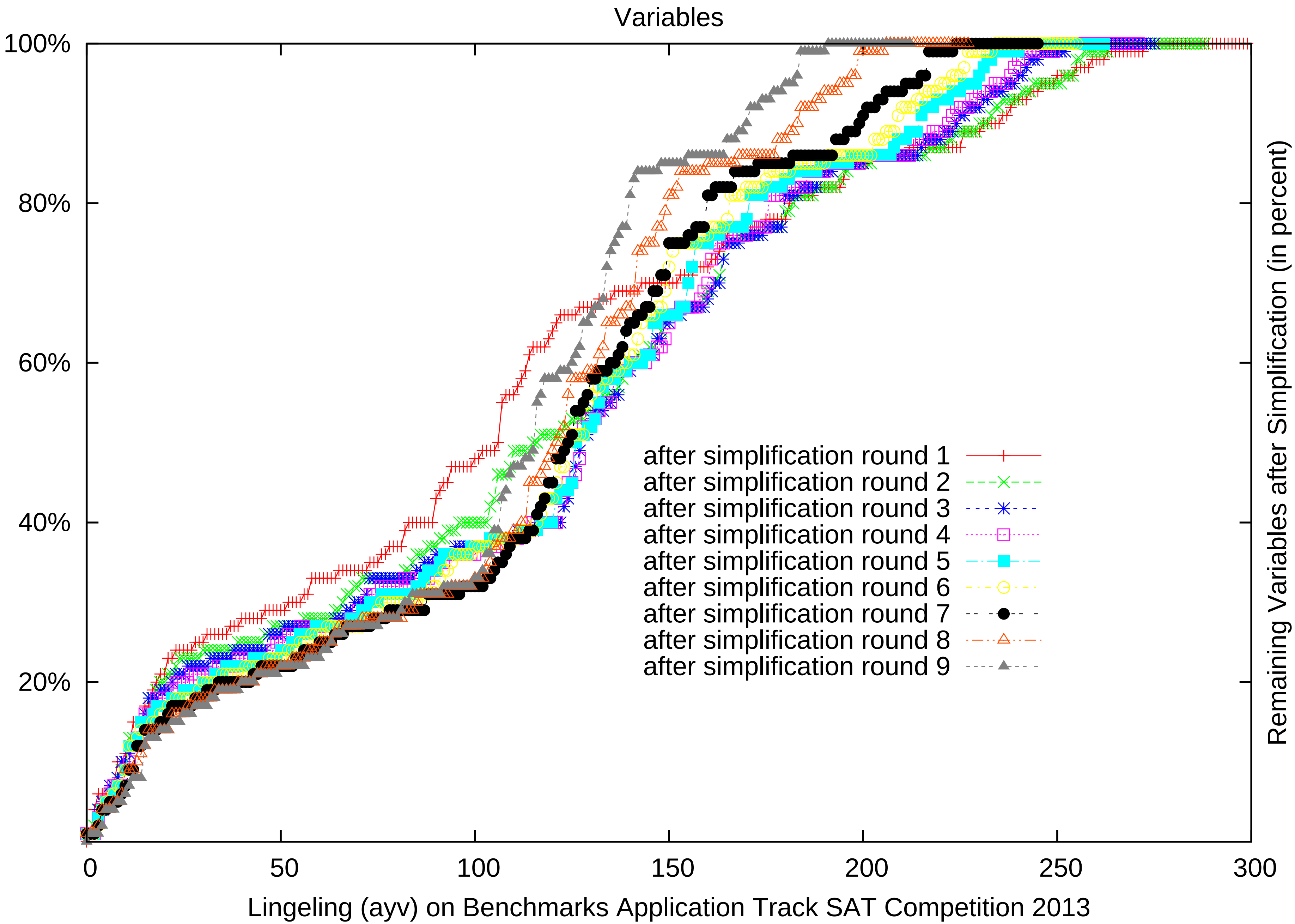

Reencoding
[MantheyHeuleBiere'HVC12]
Encoding
Inprocessing
[JärvisaloHeuleBiere'IJCAR12]
Simplifying
Search

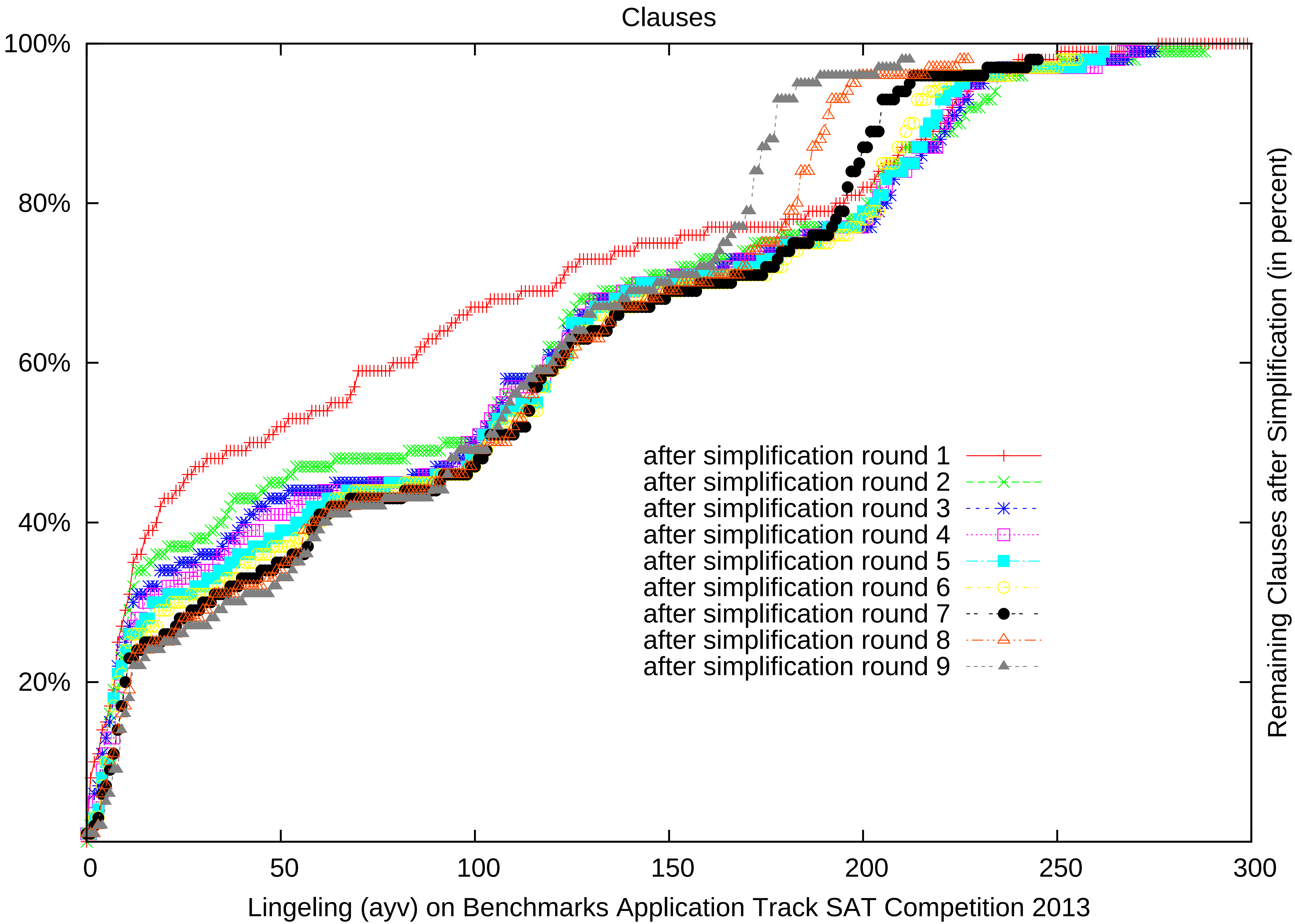
- Ternary Resolution
- Cardinality Reasoning
- Gaussian Elimination
- Equivalent Literal Substitution
- various literal probing algorithms
  - 3 variants: Root, Simple, Tree
  - + basic asymmetric tautologies (AT)
  - + lazy hyper bin resolution (LHBR)
- Congruence Closure
  - after syntactic gate extraction
- Lifting
  - double look-head probing
  - extract equivalences
  - finds units + implications
- Cliffing
  - lift units implied by literals in clause
- Unhiding
  - uses binary implication graph (BIG)
  - randomized depth first search
  - removes clauses / literals
- Transitive Reduction
  - explicit and on BIG only
- Blocked Clause Elimination (BCE)
- Covered Clause Elimination (CCE)
- Bounded Variable Elimination (BVE)
  - semantic: Minato's algorithm
  - syntactic: SatELite like
  - implicit BCE and (self) subsumption
- Blocked Clause Addition (BCA)
  - only binary clauses

some more disabled

Benefits of Inprocessing

- special case *incremental preprocessing*:
  - preprocessing during incremental SAT solving
- allows to use *costly* preprocessors
  - without increasing run-time “much” in the worst-case
  - still useful for benchmarks where these costly techniques help
  - good examples: probing and CCE — even BVE is in general costly
- additional benefit:
  - makes units / equivalences learned in search available to preprocessing
  - particularly interesting if preprocessing simulates encoding optimizations
- danger of hiding “bad” implementation though . . .
- . . . and hard(er) to debug and get right
  - our “Inprocessing Rules” IJCAR’12 paper very useful to think about what is allowed
  - need efficient testing techniques (see our TAP’13 paper on model based testing)


Variables
100%
80%
60%
40%
20%
0
50
100
150
200
250
300
Lingeling (ayv) on Benchmarks Application Track SAT Competition 2013
Remaining Variables after Simplification (in percent)
after simplification round 1
after simplification round 2
after simplification round 3
after simplification round 4
after simplification round 5
after simplification round 6
after simplification round 7
after simplification round 8
after simplification round 9



Clauses
100%
80%
60%
40%
20%
0
50
100
150
200
250
300
after simplification round 1
after simplification round 2
after simplification round 3
after simplification round 4
after simplification round 5
after simplification round 6
after simplification round 7
after simplification round 8
after simplification round 9
Lingeling (ayv) on Benchmarks Application Track SAT Competition 2013
Remaining Clauses after Simplification (in percent)

- original version scheduled inprocessing techniques individually
  - introduces restarts
  - makes it difficult to understand what is going on
  - hard to control inprocessing frequency / effort
- effort spent in phases is measured in “steps”
  - number of visited clauses for search (approx. of mems)
  - propagations for probing, resolutions for BVE etc.
  - “counters” provide deterministic execution (versus using time)
- newer versions alternate simplification and search

simplification−1 preprocessing	search−1	simplification−2 inprocessing	search−2	simplification−3 inprocessing

  - search phases limited by geometrically increasing conflict limit
  - inprocessors steps limited relative to visited clauses during search

When to Start Next Simplification Phase?

- condensed experience of 4 years tweaking inprocessing scheduling
- default simplification schedule: 0, 20k, 40k, 80k, 160k, ... conflicts
  - last conflict limit is default increment for next conflict limit
  - increment reduced relative to maximum of removed variables and clauses
    - 0% vars/clauses removed in preprocessing ⇒ 20k
    - 3% vars/clauses removed in preprocessing ⇒ 6666 = 20k / (2 + 1)
    - 9% vars/clauses removed in preprocessing ⇒ 2k = 20k / (9 + 1)
  - *as more effective inprocessing as higher its frequency*
- large reduction (of at least 5% vars/clauses removed)
  - small conflict limit increment of 2k
  - in this case increment independent of current conflict limit
- global limits
  - hard conflict limit increment of 10 million
  - soft conflict limit increment of 1 million (if at least one var / clause removed)

Which Preprocessors Should be Run?

- bounded variable elimination (BVE) most effective
  - most preprocessors "**wait**" until BVE completed once
  - exceptions in current configuration: probing, unhiding, cardinality reasoning
- similar waiting for blocked clause elimination (BCE)
  - for instance there is no point in doing CCE before BCE completed once
  - same exceptions as for BVE in current configuration
- some preprocessors can decide formula on their own
  - BVE, Gaussian elimination, cardinality reasoning, simple probing, etc.
  - those are "**boosted**" the first time they are run (given more time)
  - for instance BVE is boosted by a factor of 40x initially
- execution of an "unsuccessful" preprocessor leads to "**delay**" its next execution
  - for instance if BVE could not delete a variable skip it next time
  - this "delay" is increased with every unsuccessful attempt

Preprocessor Effort Spent in Simplification Phase

- steps (resolutions etc.) limited linearly in relation to search time (visited clauses)
  - $\mathit{Limit} = f \cdot \mathit{Visits}$ (f different for each preprocessor)
  - each preprocessor has its own "steps counter" Steps
  - requires monitoring of actual time in preprocessors (during development)
- each preprocessor has hard step limits too (like 800 million resolutions/steps in BVE)
- taking size of formula into account
  - some preprocessor require dense mode (linear in whole formula)
  - then steps limit will have size of formula as lower bound
- penalty scheme
  - unsuccessful runs increase preprocessor specific penalty P
  - large formulas size increase penalty P
  - actual steps limit divided by 2^P
- increase preprocessor internal limits for later simplifications
  - for instance limits on the number of occurrences in BVE

Undiscussed Features

- OTFS, LMTF, minimization, etc.
- internal versus external variable indices
- incremental interface: freezing, melting
- Treengeling, Plingeling
- model based testing
- callbacks, cloning
- 336 options