Journal on Satisfiability, Boolean Modeling and Computation ? (2008) ???

Cache Conscious Data Structures for Boolean Satisfiability Solvers

Geoffrey Chu and **Aaron Harwood** and **Peter J. Stuckey**
{gchu,aaron,pjs}@csse.unimelb.edu.au
NICTA Victoria Laboratory
Department of Computer Science and Software Engineering,
University of Melbourne, 3010 Australia

Abstract

Current SAT solvers are well engineered and highly efficient, and significant research effort has been put into creating data structures that can produce maximal efficiency for the core propagation engine within SAT solvers. However, there is still substantial room for improvement. As the disparity between CPU speeds and cache sizes have increased, cache conscious data structures and algorithms have become very important. They are even more important in the context of parallel SAT solving, as issues like cache contention and main memory contention can dramatically slow down a parallel SAT solver. We present a series of data structure and algorithmic modifications that are able to increase the core propagation speed of MiniSat 2.0 by an average of 80% on a set of medium sized industrial instances, and increase the speed of a parallelized version of MiniSat running with 8 threads by 140% on those same instances.

Keywords: *Boolean satisfiability, cache-aware data structures*

1. Introduction

Boolean Satisfiability solvers find a wide range of application in areas such as electronic design automation and artificial intelligence. A tremendous amount of research has been undertaken to find ways to speed up SAT solvers. Algorithmic improvements in terms of decision heuristics and learning schemes have made modern SAT solvers orders of magnitude faster than their predecessors. Besides the algorithmic improvements however, the engineering and design of data structures and the core propagation algorithm has also led to substantial increases in speed [1]. A typical modern SAT solver spends some 70-90% of its time performing Boolean constraint propagation, hence optimization of this inner propagation loop is of great importance.

SAT solving often produces highly non-local memory access patterns. Variable and clause reordering schemes have been investigated in the past [2] to try to increase locality of reference, but the non-locality is inherent in SAT solving and can never be completely eliminated.

Memory issues are even more important in parallel SAT solving and can dramatically affect the speed of a parallel SAT solver [3]. Most current multi-core systems contain L2 or L3 caches that are shared between multiple cores. When more than 1 thread is used, cache capacity contention occurs and the effective cache capacity for each thread is reduced compared to the sequential case. Cache coherency caused by false sharing can also become an issue. If the private data of different threads are mixed together within the same cache line of memory (as could be the case, for example, if each thread used malloc to allocate memory for small chunks of private data like clauses), then any of those threads writing to their private data would force all the other threads to fetch that cache line from memory again, causing an unnecessary cache miss. Main memory access on a multi-core machine is also typically slower than on single processor machines and does not scale well with the number of cores. This causes a huge problem for memory intensive algorithms such as SAT solving. These issues makes it imperative for us to design data structures and algorithms that are as cache efficient as possible.

In this paper, we modify MiniSAT 2.0 [4], a very well engineered SAT solver that came first in the industrial category of the SAT 2005 and SAT 2006 competitions, to utilise new data structures and algorithms that significantly improve its caching properties and thus its core propagation speed. We also demonstrate our improvements using our parallel solver PMiniSat, which is a simple parallelization of MiniSat 2.0.

The layout of this paper is as follows. In Section 2, we describe in detail the data structures and algorithms currently employed in MiniSat 2.0. In Section 3, we describe the data structures and algorithmic modifications we made. In Section 4, we present an experimental evaluation of our changes. In Section 5, we discuss related work. Finally in Section 6 we conclude.

2. Boolean Constraint Propagation

We begin with our SAT terminology. A *proposition* p is a Boolean variable from a universe of Boolean variables, $\mathcal{P}$. A *literal* l is either: a proposition p, or its negation $\neg p$. Define $var(p) = p$ and $var(\neg p) = p$. The *complement* of a literal l, denoted $\neg l$, is $\neg p$ if $l = p$ or p if $l = \neg p$. A *clause* C is a disjunction of literals, often treated as a set. An *assignment* is a set of literals A such that $\forall p \in \mathcal{P}.\{p, \neg p\} \not\subseteq A$, or the failed assignment $\perp$. An assignment A *satisfies* a clause C if one of the literals in C appears in A. A *theory* T is a set of clauses. An assignment is a *solution* to theory T if it satisfies each $C \in T$.

The core of any DPLL SAT solver is the Boolean constraint propagation (or unit propagation) engine that determines what literals are the consequence of a Boolean decision. Modern SAT solvers utilize a two-watched literal scheme [1] for their core propagation engine. The two-watched literal scheme represents a vast improvement over earlier counter-based schemes used in older SAT solvers, significantly cutting down on the number of memory accesses and the amount of processing required [5]. In the two-watched literal scheme, only two of the literals in any clause is being "watched" at any time. Whenever one of those watched literals is assigned a value and becomes "false" in the clause, the clause is examined to see if it is capable of causing a unit propagation.

Each watched list belongs to a literal. And each clause is attached to the watched lists of the two literals in it that are being watched. Hence the pointer to each clause exists in exactly two of the watched list vectors at any time. Whenever a variable becomes assigned, the watched list of the literal that became "false" is examined, and each of the clauses pointed to by the pointers in the watched list is examined to determine if something needs to be done.

When a clause is examined, four different things can happen:

(a) One of the literals in the clause is found to have value “true”. In this case, the clause is already satisfied and nothing needs to be done.

(b) A literal other than the other watched literal is found to be unassigned. In this case, the watched literal that just became “false” is replaced by this new unassigned literal as the watched literal, and the clause is attached to the watched list of the new watched literal.

(c) No other unassigned literal can be found, and the other watched literal is unassigned. In this case, the other watched literal is forced to have value “true” and we have a unit propagation.

(d) no other unassigned literal can be found, and the other watched literal already has value “false”. In this case we have a conflict.

Psuedo-code for Boolean constraint propagation with watch literals is shown in Figure 1. The code takes an assignment A and a decision literal d. The queue Q of literals to be processed is a FIFO queue implemented by PUSH, POP, EMPTYQ and ISEMPTY. The set $watch[l]$ is the set of clauses in T watched on literal l. $watch1[C]$ and $watch2[C]$ are the two watched literals of clause C. The algorithm maintains that if $A \cap C \neq \emptyset$, then $watch1[C] \in A$ (although this is not maintained by backjumping). We can see the four cases defined above in the **if elseif elseif else** structure, although the case (a) is split into two since the first condition is common and cheap to check.

2.1 Boolean constraint propagation in MiniSat 2.0

MiniSat 2.0 utilises a two-watched literal scheme. This is implemented in MiniSat 2.0 through a vector of “watched lists”, each of which is a vector of pointers pointing to the bodies of the clauses. Some excerpts of the MiniSat 2.0 data structure code together with their memory layout is shown in Figure 2.

The clauses have a special data structure in MiniSat 2.0. A clause consists of an initial 32 bits (`size_etc`) that contain the size of the clause as well as a few bit flags, followed by another 32 bits (`extra`) that contain the “activity” of the clause. This is followed by the literals of the clause itself, each of which take 32 bits. The clause is produced by mallocing the exact amount of memory needed to store the clause.

3. Improving the Cache behaviour

3.1 Rationale for changes

The two largest sources of memory consumption in MiniSat 2.0 are the watched lists, and the clauses. They are also the largest sources of cache misses. SAT solving typically exhibit highly non-local memory access patterns, as variable assignments can potentially require any clause to be examined, and clauses can potentially cause any variable to become fixed. Thus at any time, a very wide number of clauses and watched lists may potentially need to be examined. This makes caching highly ineffective as there is very little locality of reference.

```
bcp($A$,$d$)
    $A := A \cup \{d\}$
    $Q :=$ PUSH(EMPTYQ, $d$)
    while $\neg$ISEMPTY($Q$)
        $(Q, l) :=$ POP($Q$)
        foreach $C \in watch[\neg l]$
            if (a) $watch1[C] \in A \vee watch2[C] \in A$
                continue
            elseif (a) $\exists l' \in C \cap A$
                $watch[watch1[C]] := watch[watch1[C]] - \{C\}$
                $watch1[C] := l'$
                $watch[l'] := watch[l'] \cup \{C\}$
            elseif (b) $\exists l' \in C, \neg l' \notin A, l' \neq watch1[C], l' \neq watch2[C]$
                $watch[\neg l] := watch[\neg l] - \{C\}$
                $watch[l'] := watch[l'] \cup \{C\}$
                if $\neg l = watch1[C]$
                    $watch1[C] := l'$
                else $watch2[C] := l'$
            elseif (c) $\exists l' \in C, \neg l' \notin A$
                $watch[watch1[C]] := watch[watch1[C]] - \{C\}$
                $watch1[C] := l'$
                $watch[l'] := watch[l'] \cup \{C\}$
                $A := A \cup \{l'\}$
                $Q :=$ PUSH($Q$,$l'$)
            else (d)
                return $\perp$
    return $A$
```

Figure 1. Boolean constraint propagation of the decision literal d given a current assignment A.

For typical industrial instances where the number of clauses could number in the hundreds of thousands, cache misses can cause a substantial slowdown in the core propagation engine. It has been found that more cache aware SAT solvers like Chaff and Berkmin are faster than their predecessors by up to 3 times due to improved cache performance alone [7].

It may seem that with such awareness of cache issues and with so much work having been done to improve the core propagation engine of SAT solvers, there would be little if any improvements left that can be made. However, that turns out not to be the case. Using Cachegrind, a cache simulator in the Valgrind tool suite, we were able to identify and measure various sources of cache misses. The core of SAT solver is the inner loop of the propagation engine where a clause is examined to see if it can produce a unit propagation. This loop is run hundreds of millions of times and constitutes the bulk of the processing in a SAT solver. By running Cachegrind on 10 typical medium sized industrial instances, we were able to find out how cache misses were affecting this inner loop. It was found that the

```
class Clause {
    uint32_t size_etc;
    union { float act;
            uint32_t abst; } extra;
    Lit     data[0];
    ...
};

template<class T>
class vec {
    T*  data;
    int sz;
    int cap;
    ...
};

vec<vec<Clause*> >  watch;
```

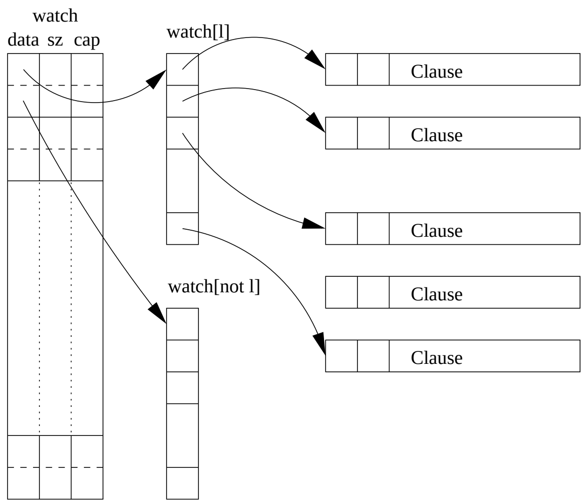

Figure 2. Data structure used in MiniSat 2.0

SAT solver spent an average of around 80 instructions, 3.2 L1 misses and 0.64 L2 misses per inner loop on the instances tested (more details are provided in the results section). Current computers have L1 miss penalties of the order of 10-15 cycles, and L2 miss penalties of the order of 100-200 cycles, thus these figures show that a very substantial portion of the run time, perhaps 60-70%, is spent simply waiting for data to be read in. Having confirmed that cache misses are indeed a major problem, we targeted each source of cache misses we found in order to design data structures that can reduce them.

In the following sections, we describe each of our modifications in turn. Our changes are aimed at two things. Firstly, to increase locality of reference by using data structures that can pack the data closely together, and second, to eliminate as much data lookup as possible.

3.2 Watched list data structure modification

The data structure currently employed by MiniSat 2.0 to store the watched lists is a "vector" data structure. A vector as implemented in MiniSat 2.0 consists of both a head and a body. The head contains the size of the vector, the amount of memory malloced for the body, and a pointer to the body. The body contains the actual elements.

Cache simulations of MiniSat 2.0 on a set of 119 medium sized industrial instance (see Appendix A) revealed that reading in the watched lists during Boolean constraint propagation causes a significant number of cache misses. The cache misses can be divided up into two groups: misses from reading the head of the watched list vector, and misses from reading the body of the watched list containing the actual data. Interestingly, the number of cache misses from reading the head of the vectors is close to the number of cache

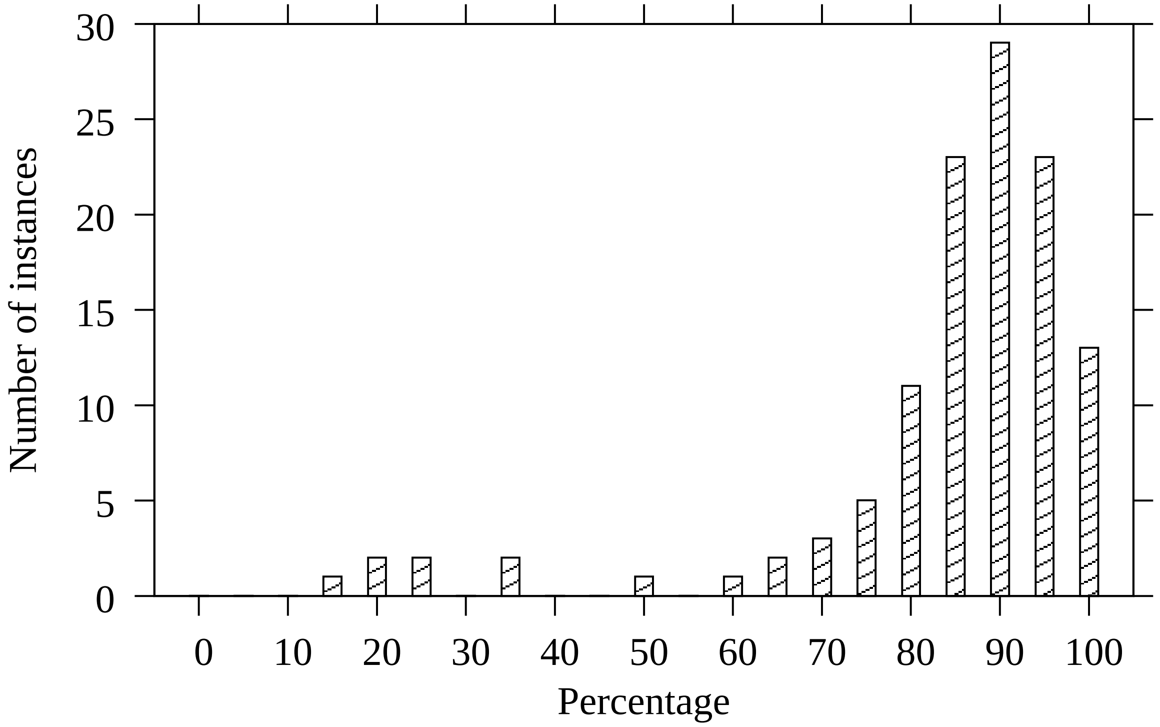

Figure 3. The histogram over the 119 tested instances of the percentage of short watched lists (length ≤ 16) encountered during the inner loop for Boolean constraint propagation.

misses from reading the body of the vectors even though the head actually contains no real information at all. This is clearly a source of inefficiency.

Further examination shows the cause of the problem. It turns out that in typical industrial instances, watched lists are often very short. The average length of the watched lists depends on the ratio between the number of clauses and the number of variables. This ratio often lies somewhere between 3-6 for the original problem depending on the problem class, and is largely independent of problem size. As more learnt clauses are created, the ratio will increase. But for a large problem, the number of learnt clauses never exceeds the number of original clauses by much. As a result, watched lists often only contain a small number of elements. For example, among the 119 problems we collected statistics for, 105 of them had over 75% of their watched lists having 16 elements or less, with most problems having around 85-90% of their watched lists being short. Further details can be found in Figure 3.

This small number of data elements means that most of the time all of the data in the watched list can be stored in just one cache line of memory. Normally, the cost of reading the vector head would be amortised over the cost of reading in the body of the vector. But since the watched list bodies often only require 1 cache line to be pulled in to read all of its data, the memory read required to read in the head becomes relatively expensive. Reading the head and the body separately basically causes 2 memory reads instead of one

```
class WatchedList {
    T*  data;
    int sz;
    int cap;
    T   head_data[12];
}
```

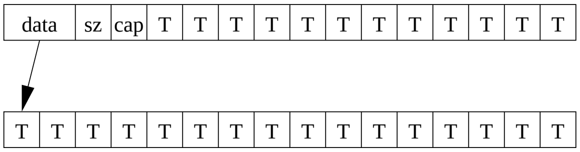

Figure 4. Watched List Data Structure

and doubles the number of reads required to read in the data and hence doubles the number of cache misses.

The vector data structure was probably chosen because the length of a watched lists can change, and so any fixed sized data structure would not work. However, this doubling of the number of memory reads is inefficient. The modification we propose is to use custom vector data structures that can store the first few elements of the watched list in the head part of the vector, and have the rest stored in the body as usual. Thus for a 64 byte cache line for example, we store the usual vector head information in the first 16 bytes, and use the remaining 48 bytes for storing actual data. This gives enough space to store 12 elements in the watched list head. The C++ definition and a diagram of the data structure is shown in Figure 4. Since watched lists are usually short, the data would often fit completely within those 48 bytes and the vector body does not need to be examined. Thus we can read the entire watched list by reading in only 1 cache line of memory. This modification basically halves the number of memory reads and cache misses caused by reading in the watched lists. However, some extra processing is required to both read and write to such a data structure.

3.3 Clause data structure modification

Examination of typical propagation behaviour also revealed some interesting statistics. It was found that clauses are often not fully examined during propagation, rather, only the first few literals are examined. Both case (a) and case (b) as described in Section 2 can result in the clause not having to be fully examined, as a true or unassigned literal can be found within the first few literals of the clause. Figure 5 shows the statistics on the distribution of how many literals of the clause are examined during each propagation. As can be seen, the probability that a clause has to be examined beyond a certain length drops very rapidly after the first few literals. In a very large proportion of the instances we investigated, some 50-90% of clause examinations terminate after just examining the first literal, and some 85-100% of clause examinations terminate within the first 4 literals.

In the original MiniSat, whenever a clause is examined, an entire cache line of data is likely to be pulled in. Since we often only actually need to read the first few literals of the clause, e.g 16 bytes out of a 64 byte cache line, pulling in an entire cache line consisting of the rest of the clause is wasteful and will tend to flush out other useful data. Furthermore, MiniSat 2.0 used a simple malloc to allocate memory for clauses, and thus we have no guarantee that clauses to not cross cache line boundaries or that there no wasted space between clauses, etc.

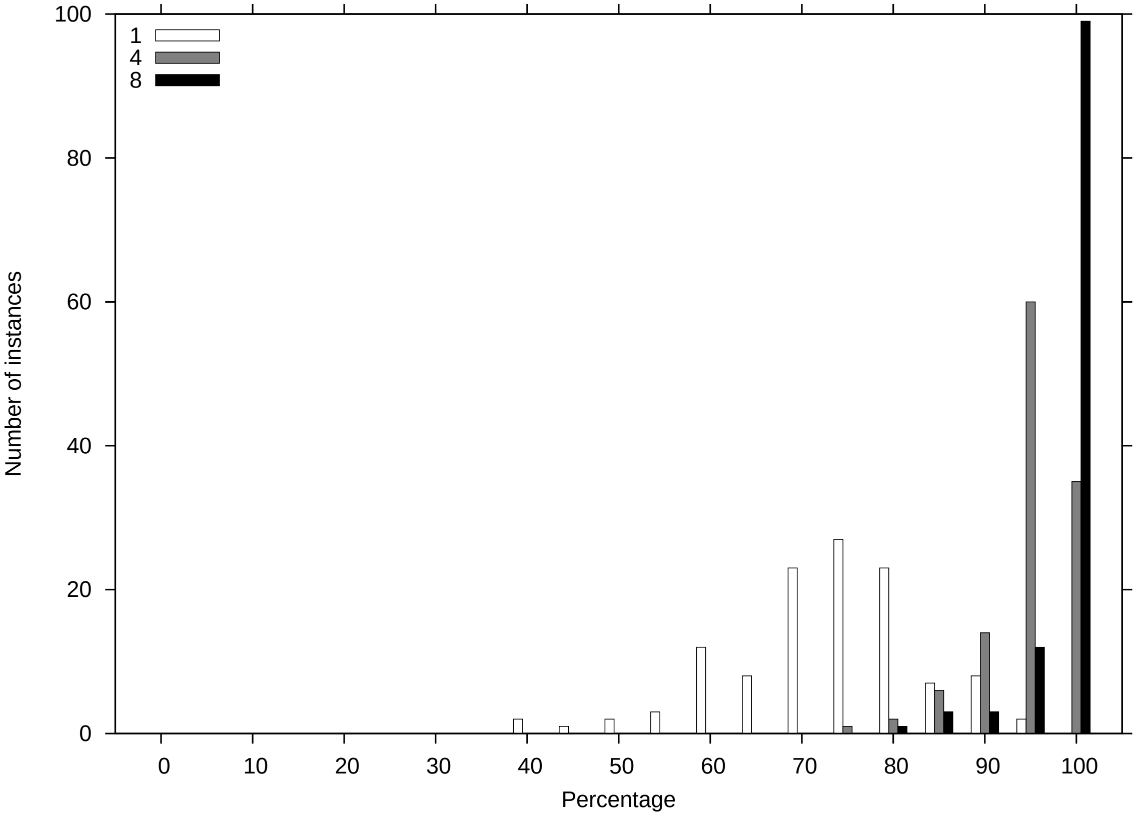

Figure 5. The histogram over the 119 tested instances of the percentage of clauses where $\leq k$ literals were examined in the inner loop for Boolean constraint propagation. The histogram shows the results for $k = 1$ (only first literal examined), $k = 4$ and $k = 8$.

The modification we propose is to split up clauses into a head part and a tail part. The head part consists of the first few literals of the clause, which is the part most likely to be examined, and the tail part consists of the rest of the literals. Since only the head part is likely to be examined in any examination of the clause, the head parts of clauses often have better locality of reference with each other than with their own tail parts. Furthermore, since we know their exact size, we can manage their memory allocation manually and ensure that they are well packed along cache line boundaries.

The optimal choice for the length of the head part varies depending on the characteristics of the problem instance. It is necessary for a whole number of the clause heads fit into each cache line, thus the available choice for the clause head sizes would for example 4, 8, 16, etc literals. If a new watch is often found within the first 4 literals in the problem instance, a choice of 4 would be best. If the instance often requires you to go beyond the first 4 literals to find a new watched literal, then a choice of 8 may be better, etc. However, in

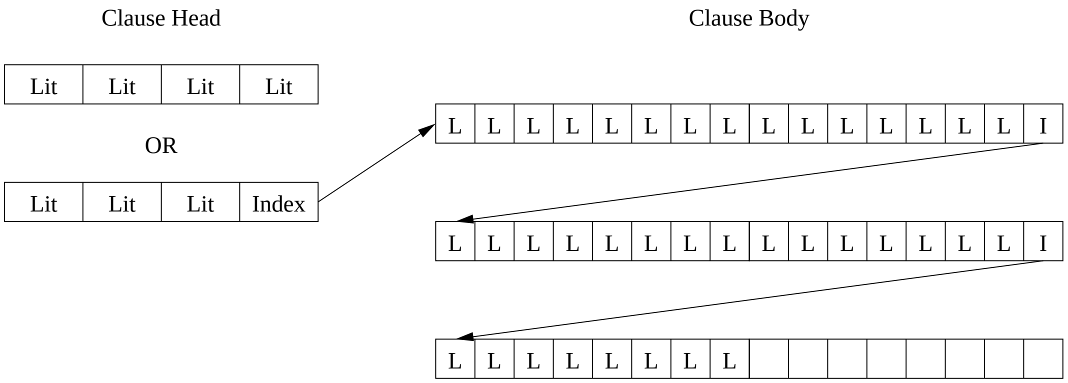

Figure 6. Clause Data Structures

general, 4 literals for the head seems to be a good choice, as the statistics shows that in most problems, some 85-100% of clause examinations terminate within the first 4 literals. Furthermore, since most of the original clauses are of length 2 or 3, picking 4 literals for the head will allow us to pack all of the original clauses very tightly together with little wasted space. Our experiments confirm that this is indeed the optimal length when this modification is implemented alone. We will later show however, that some of our other very effective modifications alters the statistics so significantly that a choice of 4 literals for the head is no longer the optimum when those other modifications are made. This will be discussed in a later section.

We implement our split clause data structure by having a vector of clause heads, and a vector of clause bodies. If a clause has length 4 or less, we simply store those 4 literals in the clause head. If the clause has length greater than 4, we store the first 3 literals in the head and store an index to the rest of the clause in the 4th spot. The body of the clause is also split up into chunks, forming a linked list where each node fits exactly in one cache line. Having fixed size chunks for both clause heads and clause bodies allows us to easily free and reuse blocks of memory that have already been allocated. There is little if any penalty for splitting up the body into chunks, as the body chunks are rarely examined, and when the end of the cache line is reached, a new cacheline would have to be read in in any case even if the clause was stored in a contiguous block of memory. A diagram of our data structure is shown in Figure 6.

A side benefit of this scheme is that the clauses can now be referred to by a 32 bit index rather than by a pointer, and on 64 bit systems, this cuts the size of the watched lists by half, further improving cache performance.

In the parallel context, since both the clause packing and the watched list modification pack the bulk of the data belonging to each thread into large contiguous arrays that belong to the thread, cache contention caused by false sharing is almost completely eliminated.

3.4 First literal

Perhaps the most significant statistic we collected is that, as noted above, in a surprisingly large proportion of time, only one literal in the clause is ever examined. It is often the case that the first literal in the clause is already assigned a value of true and thus the clause examination can immediately terminate. This occurred somewhere between 50-90% of the time on most of the instances we examined. This is due to the way that the literals in the clause are reshuffled by the propagation algorithm. i.e. literals found to have value true are moved to the first spot, and literals that are made "true" by Boolean constraint propagation on that clause are moved to the first spot. This interesting statistic means that in a significant proportion of the time, a clause is read from memory only to read the value of that first literal. This is tremendously inefficient as each such read would pull in for example a 64 byte cache line just to read 4 bytes of data.

The modification we make is to store the first literal of the clause in the watched list itself along with the index to the clause. Then when the watched list is examined to perform propagation, that literal is first checked to see if it is in fact already true, in which case the clause does not have to be read in at all. And if the literal is not true, then the index is used to read in the clause as per usual. This modification effectively eliminates the memory read required to pull in the clause some 50-90% of the time. The cost of this modification is that the size of the watched lists are now doubled. However, the locality of reference is far better on the watched lists than among randomly located clauses in memory so this modification substantially improves the caching properties of the solver.

3.5 Binary clauses

An almost trivial extension to the previous modification is to handle binary clauses specially. Since we are already storing one literal of the clause in the watched list itself, it is quite easy to have each of the literals in the binary clause store the other literal in its watched list. We can simply mark the clause index with a bit flag to indicate whether the clause is binary or not, and if it is it can be handled specially. Since knowing the other literal in the clause means that you know everything about the clause, the clause itself never needs to be read in to perform propagation.

The cache benefit of this modification is to some extent negated by the previous modification as the previous improvement has already made it unnecessary to read in a large proportion of clauses. However, experiments show that a significant enough number of binary clauses lookups got through the previous improvement to make this modification worthwhile.

3.6 Ternary clauses

A somewhat more non-trivial change is to handle ternary clauses specially as well. Since the first literal modification in Subsection 3.4 requires that we store both a 4 byte clause index and a 4 byte first literal in the watched list for each clause, there is already enough space for us to store two literals, and thus we can store all of the information contained in ternary clauses in the watched lists themselves. The different variations of the watched list element are shown in Figure 7. This change complicates certain other aspects of the solver but is certainly possible. The benefit of this modification is less clear, as although we would eliminate all lookups of ternary clauses and thus have less cache misses, each of those clauses would now be attached to three watched lists instead of two, and thus the

Normal Clause	First Literal	Clause Index
Binary Clause	Other Literal	Unused
Ternary Clause	1st Other Literal	2nd Other Literal

Figure 7. Watched list element data structure

processing required to propagate them may increase by as much as 50%. Experiments seem to indicate that there is some benefit however.

As mentioned in Subsection 3.3, the optimal choice of the clause head length depends on the statistics of the problem. For the unmodified solver, a length of 4 is optimal. However, once the first literal modification, the binary clause handling and the ternary clause handling in Subsections 3.4, 3.5 and 3.6 are implemented as well, the statistics of the problem changes significantly. All binary and ternary clause lookups are now eliminated, as well as all lookups where the first literal is already true. This represents a significant proportion of the cases where the clause examination terminated within the first 4 literals. Thus once these modifications are made it can no longer be said that most remaining clause lookups terminate within the first few literals. The new optimum was found experimentally to be 16 for our set of instances, which coincides with the size of a cache line on our machine, although the difference in speedup is only around 5-10%.

3.7 Packing assignments

Besides the above major modifications, there are also some trivial packing of data that can be done. For example, in the original MiniSat 2.0, a single byte is used to represent the current value of each variable. Because of the very non-local way in which variables are examined, it will often be the case that when the algorithm examines one variables value, an entire cache line of assignment data is pulled in, the rest of which will never be used. Since 2 bits is in fact sufficient to store a variables assignment, it is possible to pack the value of 4 variables into each byte. Such packing means that each cache line contains more assignments and there is a greater chance that some of the other assignments pulled in with the cache line has some use.

3.8 Expected benefits

Most of the above changes trade more processing for reduced cache misses. Thus they are not guaranteed to improve performance. A benefit would only be gained if the underlying assumptions for the modification hold, i.e. the source of cache misses being tackled is a major source of slow down, and the statistical assumptions behind the modification hold for the particular instance you are trying to solve. We expect this to be true for all reasonably large instances, e.g. $> 50{,}000$ variables. For small, instances however, cache misses are not an issue and the changes may well cause a slow down.

Another important point to note is that the effects of the modifications do not simply add up. In some cases there are synergies such that the total effect of the modifications is greater than the sum. In others, the effect of one modification is replicated by another to some extent and the total effect is less than the sum.

4. Experimental evaluation

4.1 Sequential solving

In the first set of experiments we test the effectiveness of our modifications in the context of sequential SAT solving using modified versions of MiniSat 2.0. An exhaustive test of each combination of data structure modifications is too time consuming. We test each modification in turn (except for the ternary clause handling as it is difficult to implement alone), all of them together, and certain combinations that produce interesting synergies.

The tests were performed on Xeon Pro 2.4GHz processors with 32kb L1 cache, 4Mb L2 cache, 64 byte cache line and 16-way set associativity. The solver was compiled using gcc 3.4.5 using -O3 optimization. 24 medium sized industrial instances from SAT competition 2007 [8] were used (See Appendix B). These instances come from 9 different problem classes. We also run our final version with all modifications on a further 33 industrial instances (see Appendix C) of varying difficulty and different problem classes to ensure that our results generalise. The original version was run 25 times with different random seeds and the modified versions were run 20 times with different random seeds. This brings the standard error down to around 5%. The instruction count and cache miss counts were collected using the Valgrind cache simulator. For the cache simulation, only 10 instances were tested with 1 run each, so these numbers are merely indicative. The results are shown in Table 1.

As the figures in Table 1 show, the modifications all increased the amount of processing required, but most of them compensated for that with reduced cache misses, leading to substantial speedups. Clause packing, when implemented alone, is most effective for a clause head length of 4, which is what we expect given the large number of binary and ternary clauses in the original problems. Clause packing is able to reduce L1 cache misses by around 10% and L2 cache misses by around 45%, resulting in a speedup of around 19%. The first literal modification also produce a significant speedup, reducing L1 cache misses by around 20% and L2 cache misses by around 40%, giving a speedup of around 25%. The case for the watched list modification, binary clause modification and assignments packing is less clear. The watched list modification is able to reduce L2 cache misses by around 20%, but the extra processing required has apparently out weighed the benefits. A more efficient implementation of the modification may yield a better speedup. The binary clause modification yielded no speedup whatsoever. In fact, it seems slightly inferior to the original. However, it produces some speedup when coupled with the first literal modification (extra 16%), as the data structure changes used in the first literal modification allow the binary clause modification to be implemented with little extra overhead. Assignments packing also yielded no speedup when implemented alone. However, as a comparison of PA+PC(4) with PC(4) shows, assignments packing can produce a non-negligible speedup (extra 11%) when coupled with other modifications. The best combination of modifications WL+PC+FL+BC+TC was able to reduce L1 misses by around 45% and reduce L2 misses by around 75%, leading to a speedup of 80%.

In summary, it can be said that the clause packing, first literal modification, and the special binary clause handling each increase the propagation speed by around 15-25%, and

Table 1. Cache conscious optimizations for sequential SAT solving. Comparison of various combinations of optimizations: WL — Watched list modification (3.2), PC(n) — Clause modification with length n clause head, n = 16 if labelled as just PC (3.3), FL — Store first literal in watched list (3.4), BC — Handle binary clauses specially (3.5), TC — Handle ternary clauses specially (3.6) PA — Pack assigns (3.7). The results show the (geometric mean) runtime for the suite, the speedup compared to MiniSat 2.0, I/Loop stands for average number of instructions per inner loop of the propagation engine, L1M/Loop and L2M/Loop stands for the average number of L1 and L2 misses per inner loop of the propagation engine.

24 medium sized industrial instances (Appendix B)

Modifications	Runtime (s)	Speedup	I/Loop	L1M/Loop	L2M/Loop
MiniSat 2.0	59.05	—	80.7	3.21	0.64
WL	62.17	0.95	97.7	3.02	0.53
BC	62.07	0.96	93.7	3.30	0.69
PA	59.17	1.00	99.7	2.89	0.62
PC	64.39	0.92	97.7	3.14	0.65
PC(8)	54.85	1.08	97.9	3.05	0.48
PC(4)	49.78	1.19	98.3	2.95	0.35
FL	47.91	1.25	83.7	2.58	0.39
PA+PC(4)	45.51	1.30	99.3	2.63	0.34
WL+PC+FL	38.96	1.53	104.5	2.44	0.29
WL+PC+FL+BC	35.14	1.70	99.3	2.23	0.24
WL+PC+FL+BC+TC	33.16	1.80	89.8	1.78	0.16

33 other industrial instances (Appendix C)

Modifications	Runtime (s)	Speedup
MiniSat 2.0	212.21	—
WL+PC+FL+BC+TC	117.77	1.80

the ternary clause handling increases it by another 10%. The changes caused by the watched list modification and the assignments packing are too inconsistent to be considered a real improvement. The average speedup gained from the best combination of modifications is around 80% on the initial 24 instances we examined. There is a large amount of individual variation in the speedups of the instances, with the speedup ranging between 19% to as high as 178%. The average speedup on the 33 extra instances we tested it on is also 80%. Thus it is quite likely that the speedup generalises to other similarly sized instances.

4.2 Parallel solving

In the second set of experiments we test the effectiveness of our modifications in the context of parallel SAT solving using the parallel SAT solver PMiniSat. Two versions of PMiniSat were made, one with and one without the data structure modifications. The tests were

Table 2. Cache conscious optimizations for parallel SAT solving. We compare a parallelized version of MiniSat2.0, PMiniSat, using the original data structures and the combination of optimizations detailed in Table 1. The results are the geometric mean of 20 runs.

	Original		WL+PC+FL+BC+TC		
Threads	Runtime (s)	Efficiency	Runtime (s)	Efficiency	Speedup
1	44.03	—	28.30	—	1.56
2	26.94	0.82	15.19	0.93	1.77
3	21.75	0.67	12.70	0.74	1.71
4	19.06	0.58	10.52	0.67	1.81
5	19.90	0.44	10.15	0.56	1.96
6	21.42	0.34	10.70	0.44	2.00
7	23.41	0.27	10.31	0.39	2.27
8	26.44	0.21	11.03	0.32	2.40

performed on an 8 core Mac with 2x Harpertown Quad core Xeon 2.8GHz processors. Each core has a 32kb L1 cache, and each pair of cores share a 6Mb L2 cache with a 64 byte cache line and 24-way set associativity. The solver was compiled using gcc 4.0.1 using -O3 optimization. We run both versions of the solver with 1 to 8 threads. Again the tests are done on the 24 medium sized industrial instances (Appendix B). Each instance was run 20 times with different random seeds. The results are given in Table 2.

The sequential (1 thread) speedup due to the modifications is slightly lower on the Mac than on the Xeon Pro. This is to be expected as the solver has access to a full 6Mb of L2 cache on the Mac compared to only 4Mb on the Xeon Pro, thus cache misses are less of an issue and the cache improvements have a smaller effect. The speedups produced by the modifications clearly increase with increasing number of threads, from 56% speedup for 1 thread up to 140% for 8 threads.

The dramatic drop in efficiency as more threads are used is somewhat surprising at first glance, as it is known from other experiments that PMiniSat is highly efficient algorithmically, achieving an efficiency of 103% at 4 threads and 87% at 8 threads on these 24 instances in terms of the total number of executions of the inner loop over all threads. The communication cost in PMiniSat is also negligible. Further investigation shows that the relatively low efficiency of PMiniSat on this multi-core machine (with or without cache improvements) can be directly attributed to memory costs.

Our cache simulation numbers show that the average rate of L2 cache misses on the unmodified sequential solver was of the order of 10 million misses per second. With a write-back cache policy, and a 64 byte cache line size, this translates into a fairly massive 1.28 Gb/s of traffic between the processor and the main memory. Thus even though the entire clause database of a medium sized industrial problem may only require 20-50 Mb of memory, the highly non-local memory access pattern forces an enormous amount of memory transfer to occur. The situation is far worse for our parallel solver, as the memory bandwidth requirement will be multiplied by the number of threads. Futhermore, issues such as cache contention, shared data and speculative prefetching by the Harpertown processors

Table 3. Parallel memory stress test.

Threads	Runtime (s)	Efficiency
1	29.2	—
2	22.6	0.65
4	18.2	0.40
8	16.8	0.22

will increase the amount of memory transfer that each thread needs even further. In the sequential solver, L2 misses were already consuming some 50-60% of the cpu time. From our numbers, it appears that in the parallel solver, this proportion grows to around 75% for 4 threads and around 90% for 8 threads. Thus when the parallel solver is run with many threads, the limiting factor is not processing power, but the ability of the main memory to handle an extremely large number of memory accesses. On the 8-core Mac that the experiments were run on, the main memory is clearly not fast enough to satisfy all 8 cores.

To illustrate this, memory stress tests using a memory access pattern similar to a SAT solver were run on the 8 core Mac. In the stress tests, 64 Mb of contiguous memory is allocated to each thread. Each thread then reads a random location within its block and increments the value. This loop is performed a billion times. As can be seen in table 3, the efficiency drops dramatically with increasing number of threads and the trend is very similar to what is observed on the parallel SAT solver.

In the sequential context, our modifications were able to substantially reduce cache misses. In the parallel context, they provide even more speedup because of the other memory issues that occur. When multiple threads share an L2 cache, capacity contention occurs. Since the vast majority of cache misses in SAT solving are capacity misses, when two threads share the same cache, the effective L2 cache size for each of them is roughly halved. On the Harpertown processors, each thread would effectively have 3Mb or less of L2 cache instead of 6Mb for the sequential case. Thus L2 misses are far higher and our cache improvements produce more speedup. Main memory contention is clearly a very serious source of inefficiency for parallel SAT solvers. Our cache improvements reduce L2 misses, which translates into less main memory accesses and thus less contention, once again producing more speedup. Although our numbers show that it would be impractical to run our solver with more than 4 or 5 threads on this particular machine, the speedups illustrate very well how much more effective our modifications are in the parallel context, as our modifications produce a far greater speedup of 140% with 8 threads than the 56% with 1 thread.

5. Related work

The inefficiency of having the clauses reside in non-contiguous, randomly allocated memory locations has been noted before, and array style clause memory management have been attempted. For example, in Chaff [1], a large chunk of contiguous memory is initially allocated for clauses. Memory from this pool is then allocated to clauses as needed. However, in their implementation, the memory is simply allocated from the end of the pool, and they do not attempt to reuse freed chunks of memory caused by deleted clauses until an infrequent monolithic garbage collection step compactifies the remaining clauses. This somewhat reduces the effectiveness of using an array for the clauses, as clauses are no longer in contiguous parts of memory due to there being deleted clauses in between. Our implementation allocates fixed sized chunks of memory to clauses, thus it is easy for us to keep a stack of freed chunks that can be reused immediately. This makes better use of memory and also produces better locality of reference.

Our first literal modification bears some resemblence to the Watched Literal Reference List (WLRL) used in [9], however, their WLRL data structure is motivated by different considerations. In their parallel solver MiraXT, clauses are read-only and are shared among multiple threads, thus in order for watched literal propagation to occur, each thread has to remember the two literals in the clause that are attached to their watched lists. This data is stored in an extra data structure called the WLRL. Similar to our first literal modification, examining the WLRL can allow the clause examination to terminate without examining the original clause. However, the authors do not examine the cache behaviour of this design or whether it can produce any speedups.

The clause data structures used in CMUSAT [10] is also similar to our first literal modification and is motivated by the same reasons. However, instead of storing the first literal in the watched list itself as we do, they store the two watched literals of each clause in a separate array, essentially creating an abridged version of each clause. This is very similar to the WLRL described above. When a clause is examined, their algorithm first examines these two watched literals to see if they are true. If they are, then the main clause is not examined. It seems to us however, that such a design may not decrease cache misses by much. The examination of the two watched literals, which is stored in a separate array from the watched list, is very likely to cause a cache miss as well, as it has the same highly non-local access pattern that the clauses suffer from. And in the case that the main clause has to be examined, their design may end up causing two misses instead of one. Our modification is superior, as storing the first literal in the watched list itself provides far better locality of reference.

The special handling of binary clauses is not a new idea. In fact, MiniSat 1.13 had a similar implementation that was later removed in MiniSat 2.0 for readability reasons. Special handling for ternary clauses have also been considered before, for example [6] discusses ways in which ternary clause could be handled specially by packing the literals together. Their packing scheme involves packing the 3 literals into only 64 bits. This can substantially reduce the memory footprint when a large majority of clauses are of length 3. However, their implementation limits each literal to 20 bits, with 1 bit reserved for a flag. This means that a maximum of 524,288 variables are allowed. Such a restriction may have been reasonable at the time, but is no longer really reasonable given the sizes of the problems that can be solved today. Our changes do not modify the length of a literal, but do utilise two of its bits for special flags. Thus we retain a limit of 536,870,912 variables, which ought to be sufficient for the foreseeable future.

6. Conclusion

This paper describes a series of data structure and algorithmic modifications that are able to increase the core propagation speed of MiniSat 2.0 by some 80% on a wide range of medium sized industrial instances. An even greater speedup of around 140% was obtained when the modifications were incorporated into the parallel SAT solver PMiniSat running with 8 threads. Cache simulations were used to find the main sources of cache misses in MiniSat 2.0. Propagation statistics were then collected that guided us towards the design of more cache concious data structures and algorithms. Three of our modifications produced unambiguous speedups of the order of 15-25% each in the sequential context. Our experiments indicate that memory costs dominate execution in the parallel solver, and hence cache improvements are even more important. Our modifications are relatively easy to implement and can easily be incorporated into other state of the art SAT solvers, all of which currently also utilise the same two watched propagation as MiniSat 2.0.

References

[1] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an Efficient SAT Solver. *Proceedings of the 38th Conference on Design Automation*, pages 530–535 (2001)

[2] F. A. Aloul, I. L. Markov, K. A. Sakallah, FORCE: A Fast and Easy-To-Implement Variable-Ordering Heuristic. *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 116–119 (2003)

[3] Y. Feldman, N. Derschowitz and Z. Hanna, Parallel Multithreaded Satisfiability Solver: Design and Implementation. *PDMC* (2004)

[4] N. Een, N. Sörensson, MiniSat - A SAT Solver with Conflict-Clause Minimization. *Proc. Theory and Applications of Satisfiability Testing (SAT'05)* (2005)

[5] I. Lynce, J. Marques-Silva, Efficient data structures for backtrack search SAT solvers. *Proc., Fifth International Conference on Theory and Applications of Satisfiability Testing (SAT'02)* (2002)

[6] L. Ryan, Efficient Algorithms for Clause-Learning SAT Solvers. *M.Sc. Thesis*, simon Fraser University, (2004)

[7] L. Zhang and S. Malik, Cache performance of SAT solvers: A case study for efficient implementation of algorithms. *Proc., Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, pages 287–298 (2003)

[8] SAT 2007 Competition. *Online. http://www.satcompetition.org/2007/*

[9] M. Lewis, T. Schubert and B. Becker, Multithreaded SAT Solving. *In 12th Asia and South Pacific Design Automation Conference* (2007)

[10] H. Jain, E. Clarke, SAT Solver Descriptions: CMUSAT-Base and CMUSAT. *In SAT competition 2007* (2007)

Appendix A. 119 industrial SAT instances

These examples were taken from the SAT Race 2008 instances available at `http://www-sr.informatik.uni-tuebingen.de/sat-race-2008/downloads.html`.

```
aloul-sr06-chnl10-13.cnf
AProVE07-04.cnf
AProVE07-06.cnf
AProVE07-15.cnf
AProVE07-20.cnf
AProVE07-22.cnf
blocks-4-ipc5-h21-unknown.cnf
clauses-10.cnf
clauses-4.cnf
cube-9-h10-unsat.cnf
cube-9-h11-sat.cnf
dated-10-15-u.cnf
dated-5-11-u.cnf
een-pico-sr06-pr02-02.cnf
een-tipb-sr06-par1.cnf
een-tipb-sr06-tc6b.cnf
eq.atree.braun.8.unsat.cnf
eq.atree.braun.9.unsat.cnf
goldb-heqc-desmul.cnf
goldb-heqc-i8mul.cnf
goldb-heqc-k2mul.cnf
goldb-heqc-rotmul.cnf
grieu-vmpc-s05-05s.cnf
grieu-vmpc-s05-24s.cnf
grieu-vmpc-s05-27r.cnf
hoons-vbmc-s04-05.cnf
hoons-vbmc-s04-06.cnf
hoons-vbmc-s04-07.cnf
hoons-vbmc-s04-07.cnf
manol-pipe-c10b.cnf
manol-pipe-c10bi_s.cnf
manol-pipe-c10id_s.cnf
manol-pipe-c10ni_s.cnf
manol-pipe-c10nid_s.cnf
manol-pipe-c10nidw_s.cnf
manol-pipe-c6b_i.cnf
manol-pipe-c6bid_i.cnf
manol-pipe-c6id.cnf
manol-pipe-c6idw_s.cnf
manol-pipe-c6n.cnf
manol-pipe-c6ni_s.cnf
manol-pipe-c6nid_s.cnf
manol-pipe-c6nidw.cnf
manol-pipe-c6nidw_i.cnf
manol-pipe-c7_i.cnf
manol-pipe-c7_i.cnf
```

```
manol-pipe-c7idw.cnf
manol-pipe-c7idw_s.cnf
manol-pipe-c8_i.cnf
manol-pipe-c8b_i.cnf
manol-pipe-c8b_i.cnf
manol-pipe-c8n.cnf
manol-pipe-cha05-113.cnf
manol-pipe-cha05-143.cnf
manol-pipe-f6b.cnf
manol-pipe-f6i.cnf
manol-pipe-f6n.cnf
manol-pipe-f6nid.cnf
manol-pipe-f7idw.cnf
manol-pipe-g10idw.cnf
manol-pipe-g6bid.cnf
manol-pipe-g7n.cnf
manol-pipe-g8b.cnf
manol-pipe-g8bidw.cnf
manol-pipe-g8n.cnf
miza-sr06-md5-47-03.cnf
miza-sr06-md5-48-01.cnf
miza-sr06-sha0-35-03.cnf
mizh-md5-47-3.cnf
mizh-sha0-35-2.cnf
mizh-sha0-35-4.cnf
mizh-sha0-35-5.cnf
narai-vpn-10s.cnf
narai-vpn-sat05-02s.cnf
narai-vpn-sat05-07.cnf
schup-l2s-s04-abp4.cnf
schup-l2s-s04s2-09.cnf
schup-l2s-s04-valves.cnf
simon-mixed-s02bis-01.cnf
simon-mixed-s02bis-03.cnf
simon-mixed-s02bis-05.cnf
simon-mixed-s02bis-05.cnf
stric-bmc-ibm-10.cnf
stric-bmc-ibm-12.cnf
total-10-11-u.cnf
total-5-11-u.cnf
total-5-13-u.cnf
total-5-17-s.cnf
uts-l05-ipc5-h26-unsat.cnf
uts-l05-ipc5-h27-unknown.cnf
uts-l06-ipc5-h28-unknown.cnf
uts-l06-ipc5-h29-unknown.cnf
uts-l06-ipc5-h30-unknown.cnf
vange-color-inc-54.cnf
velev-eng-uns-1.0-04.cnf
velev-eng-uns-1.0-04a.cnf
velev-fvp-sat-3.0-12.cnf
```

```
velev-live-sat-1.0-01.cnf
velev-live-sat-1.0-03.cnf
velev-npe-1.0-02.cnf
velev-npe-1.0-03.cnf
velev-pipe-1.0-08.cnf
velev-pipe-1.0-09.cnf
velev-pipe-1.1-03.cnf
velev-pipe-1.1-05.cnf
velev-pipe-uns-1.0-08.cnf
velev-pipe-uns-1.0-14.cnf
velev-pipe-uns-1.1-05.cnf
velev-sss-1.0-05.cnf
velev-sss-1.0-cl.cnf
velev-vliw-sat-2.0-02.cnf
velev-vliw-sat-2.0-04.cnf
velev-vliw-uns-2.0-02.cnf
vmpc_24.cnf
vmpc_25.cnf
vmpc_26.cnf
vmpc_27.cnf
vmpc_28.cnf
```

Appendix B. 24 medium sized industrial instances

```
AProVE07-04.cnf
AProVE07-06.cnf
AProVE07-15.cnf
AProVE07-22.cnf
blocks-4-ipc5-h21-unknown.cnf
clauses-4.cnf
cube-9-h10-unsat.cnf
cube-9-h11-sat.cnf
dated-10-15-u.cnf
dated-5-11-u.cnf
manol-pipe-c10nidw_s.cnf
manol-pipe-c6nidw_i.cnf
mizh-md5-47-3.cnf
mizh-sha0-35-2.cnf
mizh-sha0-35-4.cnf
mizh-sha0-35-5.cnf
total-10-11-u.cnf
total-5-11-u.cnf
total-5-13-u.cnf
uts-l05-ipc5-h26-unsat.cnf
uts-l05-ipc5-h27-unknown.cnf
uts-l06-ipc5-h28-unknown.cnf
uts-l06-ipc5-h29-unknown.cnf
uts-l06-ipc5-h30-unknown.cnf
```

Appendix C. 33 more industrial instances

```
manol-pipe-c10ni_s.cnf
manol-pipe-c6id.cnf
manol-pipe-c6nid_s.cnf
manol-pipe-c7idw.cnf
manol-pipe-c8n.cnf
manol-pipe-g10idw.cnf
narai-vpn-10s.cnf
vange-color-inc-54.cnf
velev-live-sat-1.0-01.cnf
velev-live-sat-1.0-03.cnf
velev-npe-1.0-02.cnf
AProVE07-02.cnf
AProVE07-03.cnf
AProVE07-06.cnf
AProVE07-08.cnf
AProVE07-09.cnf
AProVE07-16.cnf
AProVE07-21.cnf
AProVE07-27.cnf
blocks-4-ipc5-h22-unknown.cnf
dated-10-11-u.cnf
dated-10-13-u.cnf
dated-5-15-u.cnf
dated-5-17-u.cnf
manol-pipe-c10nidw_s.cnf
manol-pipe-c7bidw_i.cnf
manol-pipe-c7nidw.cnf
manol-pipe-f9b.cnf
manol-pipe-g10bidw.cnf
total-10-13-u.cnf
uts-l06-ipc5-h30-unknown.cnf
uts-l06-ipc5-h31-unknown.cnf
uts-l06-ipc5-h32-unknown.cnf
```