MiraXT '08 – Solver Description

Tobias Schubert Matthew Lewis Bernd Becker
Institute of Computer Science, Albert-Ludwigs-University of Freiburg, Germany
{schubert, lewis, becker}@informatik.uni-freiburg.de

Abstract

This paper briefly describes MiraXT, a multi-threaded SAT solver that was designed to take advantage of current and future shared memory multiprocessor systems. The experimental results in [2] show that already in single-threaded mode, MiraXT compares well to other state-of-the-art solvers on a wide range of industrial problems. In threaded mode, it provides cutting edge performance, as speedup is obtained on both satisfiable and unsatisfiable instances. The paper highlights some of the design and implementation details that allow multiple threads to run and cooperate efficiently.

1. MiraXT

1.1. Overview

MiraXT is a significantly enhanced re-implementation of MIRA [1] that is able to run with multiple threads. It contains MIRA's *Early Conflict Detection BCP* as well as *Implication Queue Sorting*. As the decision strategy, a modified VSIDS algorithm is used, in which all scores over 1024 are concatenated so that a bucket sort can be used to sort the list in linear time with respect to the number of variables. This allows us to sort the list more frequently keeping it up-to-date, and makes the decision heuristic less greedy. Furthermore, a preprocessing unit has been incorporated into MiraXT, that is similar to the one integrated in MiniSat2. Lastly, our approach was implemented in C++ using POSIX threads. The overall design of MiraXT is given in Figure 1.

1.2. Shared Clause Database

MiraXT uses one master clause database, the so called *Shared Clause Database*, that stores pointers to the original problem clauses, plus pointers to all the conflict clauses generated by each thread. Each clause is only present once in memory, and is shared between all threads. In order to insure coherency within the database, a lock must be acquired before a thread inserts a pointer to its newly generated conflict clause. As soon as the pointer is inserted and the database clause counter is incremented the lock is released. All clauses, once generated, are read-only, so that sharing can be done without locks. These steps are important as we want to reduce the amount of locks needed by the solver, and remove any lock contention and wait times that might result from the remaining locks. Also, each thread has one lock associated with it that is used when the thread requests a new clause from the shared clause database. This lock is used to increment its current database position pointer. This pointer keeps track of which clauses the thread has already looked at, and those that can still be added.

Clause deletion is also an important issue. In MiraXT, each thread deletes clauses using an algorithm in which inactive clauses are easier to delete than active ones. To facilitate clause deletion efficiently in our approach, each thread has one Boolean variable associated with it for every clause. Also, each clause consists of an array of literals with the first few spots in the array being reserved. These spots specify the clause length and its unique database reference number. When a thread deletes its references to a clause, it must set its Boolean variable for that clause using the clauses reference number. Because the Boolean variable for the clause is specific for that thread, no global lock is required when deleting clauses.

Once a thread has deleted all the clauses it wants to delete, it will ask the shared clause database to see if a *global* clause deletion procedure should be run, as the threads only delete their references to clauses, and not the actual clauses themselves. In MiraXT, a simple test based on how many threads there are, and how many deletion processes have been run, is used to decide if such an operation is required. If the shared clause database needs cleaning, the thread grabs a lock and proceeds to delete clauses that are no longer used by any thread, relinquishing the lock when it is finished. This lock is used to insure that no two threads run a clause deletion operation on the shared clause database at the same time.

Using the fine grained lock system described above, practically all lock contention issues were removed, and in testing we saw no signs of even light lock contention.

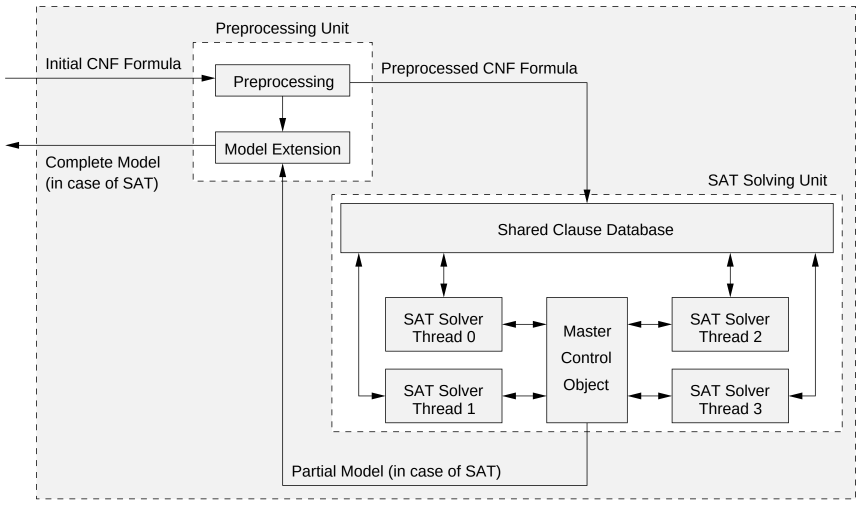

Figure 1. Overall design of MiraXT

1.3. Watched Literals Reference List

In most solvers, to keep track of the watched literals, the original clause is modified in some way (e.g., by using the first two literals in the clause). This is not possible in MiraXT, because clauses are read-only. So, each thread creates a second data structure called the *Watched Literals Reference List* (WLRL). For each clause, this structure contains two watched literals and a up to two additional literals, that are checked first by the BCP procedure when searching for a new watched literal. The WLRL basically allows each thread in MiraXT to have a condensed reference or copy of every clause. In experiments on a selection of BMC problems we observed, that 84% of the clauses can be directly evaluated with only the WLRL. This means the original clauses are not needed 84% of the time! Also, on many problems, clauses with 3 literals or less are fairly common, meaning the entire clause can be stored here. In any case, this allows MiraXT to better utilize each CPU's cache memory.

1.4. Multithreaded Solver Control

MiraXT contains no controlling master process unlike many other parallel solvers. Instead, a master control object (MCO) allows the threads to communicate with each other. All communication is done in a passive way, such that the MCO will not interfere with the threads. It will only store messages and suspend threads which ask for it to do so. The solver threads poll the MCO occasionally to check for messages, such as idle threads waiting for a new subproblem.

1.5. Multithreaded Conflict-Driven Learning

The conflict analysis procedure in MiraXT is based on zChaff's *1UIP* scheme. However, a separate clause addition procedure was implemented. In MiraXT, the conflict analysis procedure will add a clause pointer to the shared clause database. Then the clause addition procedure will be run, asking the shared clause database for all new clauses the particular thread hasn't looked at so far; this includes clauses generated by other threads and the thread's newly generated conflict clause. It will then process these clauses, deciding which clauses should be added. Currently, all conflict clauses, undefined clauses, or really short ones (10 literals or less), are added. The clause addition procedure will assign watched literals, search for implications, and perform conflict driven backtracking as needed. Both the conflict analysis procedure and the clause addition procedure can signal that the current subproblem is unsatisfiable.

2. SAT Race 2008

With respect to last year's version, we have slightly changed the decision heuristic and the clause deletion mechanism. Additionally, special BCP routines for binary and ternary clauses as well as for clauses containing exactly 4 literals have been integrated.

References

[1] M. Lewis, T. Schubert, and B. Becker. Speedup Techniques Utilized in Modern SAT Solvers – An Analysis in the MIRA Environment. In *8th International Conference on Theory and Applications of Satisfiability Testing*, 2005.

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