Open access • Proceedings Article • DOI:10.1109/ASPDAC.2007.358108

# Multithreaded SAT Solving — Source link

Matthew Lewis, Tobias Schubert, Bernd Becker

**Institutions:** University of Freiburg

**Published on:** 23 Jan 2007 - Asia and South Pacific Design Automation Conference

**Topics:** Distributed memory, Shared memory, Distributed shared memory, Speedup and Boolean satisfiability problem

# Multithreaded SAT Solving

Matthew Lewis, Tobias Schubert, Bernd Becker
Institute for Computer Science
Albert-Ludwigs-University of Freiburg,
Georges-Koehler-Allee 51, 79110 Freiburg, Germany
{lewis,schubert,becker}@informatik.uni-freiburg.de
http://ira.informatik.uni-freiburg.de

**Abstract - This paper describes the multithreaded MiraXT SAT Solver which was designed to take advantage of current and future shared memory multiprocessor systems. The paper highlights design and implementation details that allow the multiple threads to run and cooperate efficiently. Results show that in single threaded mode, MiraXT compares well to other state of the art solvers on Industrial problems. In threaded mode, it provides cutting edge performance, as speedup is obtained on both SAT and UNSAT instances.**

## I. Introduction

The Boolean Satisfiability (SAT) solvers of today, are considerably more advanced than the original Davis-Putnam algorithm [25]. Many performance enhancements have been more algorithmic, such as Non-Chronological Backtracking with Conflict Clause Learning [3,5], and novel Decision Strategies (VSIDS [1], BerkMin [2], and VMTF [8]). However, many changes, especially from zChaff, improved the implementation of these algorithms by considering the hardware that was running the solver. For example, zChaff introduced watched literal lists [4], that effectively use the caches of modern CPUs. Of course, the algorithm as a whole was also implemented efficiently, allowing zChaff to get more out of each CPU cycle.

Since Moore's prediction approximately 40 years ago [28], chip manufacturers have been doubling the number of transistors on a chip roughly every two years. Recently, new processes have given chip designers an overabundance of free transistors. To utilize all these transistors, multicored and multithreaded CPUs were introduced. In the x86 world, Intel started by adding Hyper-Threading, in which one CPU can run two threads simultaneously, sharing the CPU's internal resources. Now both AMD and Intel have taken the next step with their X2 and Pentium 4 D lines which contain two physical CPU's on one die, or in one package. This trend will continue in the future, providing CPU's with 4 or more cores. Some higher end CPUs such as SUN's UltraSPARC T1 processor (8 cores, 32 threads), or IBM's POWER5 Quad-MCM (4 cores, 8 threads) have already done this.

Basically, future SAT solvers will be running on shared memory multi-CPU systems. Work has been done on parallelizing SAT solvers for use on asynchronous distributed systems, using some form of message passing. Such examples are GridSAT [19], PaSAT [21], PaMIRA [23], and others [16,17,18,20,22,24]. Message passing, however, is slow and requires a lot of overhead when compared to a well designed shared memory system. Recently, work has been published on a multithreaded shared memory solver called ySat [15]. This paper concluded that these types of solvers have a detrimental effect on cache performance, thus degrading the overall performance of the entire solver. On the contrary, we will show that a well designed multithreaded shared memory solver can provide speedup on many industrial benchmarks.

The following section will start with an overview of the SAT problem, then describing how SAT solvers works. The shared memory multiprocessor system used is described in Section III. Next, our solver MiraXT will be discussed, highlighting single and multithreaded performance features and optimizations. Then experimental results will be shown followed by a few closing remarks.

## II. SAT and Parallel SAT

In many different research fields from Verification to Artificial Intelligence, problems can be described as a Boolean Satisfiability Problem and formatted in Conjunctive Normal Form (CNF). This consists of a conjunction of clauses, with each clause consisting of the inclusive disjunction of literals. A literal is the occurrence of a variable in its positive or negative form. A SAT solvers's task is to find a solution to the problem such that the entire formula evaluates to 1 or to prove that no solution exists.

$$F(x_1,..,x_n)=(\ x_1+x_2)\cdot(\ x_1+\ x_2+x_3)\cdot(\ x_1+\ x_2+\ x_3)\cdot\ldots$$

A single threaded SAT solver starts with all the variables in an undefined state. Then, using a heuristic, a decision is made assigning a variable to a value (1 or 0). Such a variable is called a decision variable. After every decision, a Boolean Constraint Propagation (BCP) procedure is run to find implications resulting from that decision. Most solvers maintain a chronological list of decision variables and the implication found by the BCP procedure in a decision stack. Each decision and resulting implications are referred to as a decision level, with the first decision and its implications on level 1. Decision level 0 however, contains implications that do not depend on a decision (e.g. implications from so called unit clauses). As the BCP procedure runs, it can also find conflicts, evoking a conflict analysis procedure to find the reason for conflict. This procedure would then try to resolve the conflict by backtracking to a previous decision level. It would also record a conflict clause to prevent the conflict from being repeated. If the conflict analysis procedure finds a conflict on decision level 0, the problem is unsatisfiable. Otherwise, if the BCP procedure finishes and no conflicts are found and all variables are defined, the problem is solved. For an in-depth overview of a modern SAT solver, refer to [6].

1-4244-0630-7/07/$20.00 ©2007 IEEE.

In a parallel SAT solver, each thread or process operates in the same way; however there are a few points that should be highlighted. First, conflict clauses can be exchanged between parallel running solvers. This exchange of knowledge allows all solvers to benefit from what each other has learnt. Secondly, for a parallel SAT solver, the problem space must be divided. The most obvious way is to use decision variables, as both possible values of every decision variable must be searched in order to prove unsatisfiability. Normally, the chronologically first decision variable is chosen. Fig.1 shows how a solver with two threads could operate. Once the search space is divided, both solver processes can operate like normal single threaded solvers. If one solver finds a solution, the search is over. However, if one solver proves its half is unsatisfiable, the remaining subproblem from another solver can be re-divided in the same manner. This method of dividing the decision stack is referred to as the guiding path method by PSATO [24].

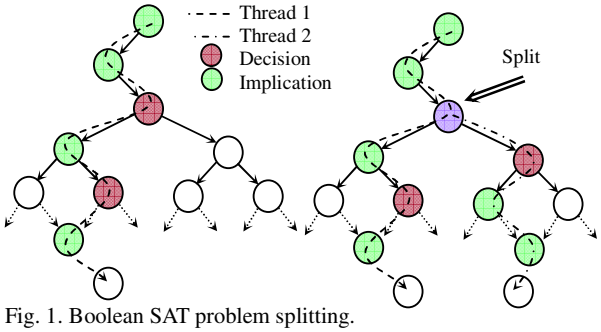


Fig. 1. Boolean SAT problem splitting.

## III. Multiprocessor System and Solver Designs

### A. AMD Opteron System

This section will cover the AMD Opteron shared memory system used in the experiments in this paper, providing a quick overview of the hardware so that the reader can better understand the following sections, discussing the optimizations made to the threaded solver MiraXT.

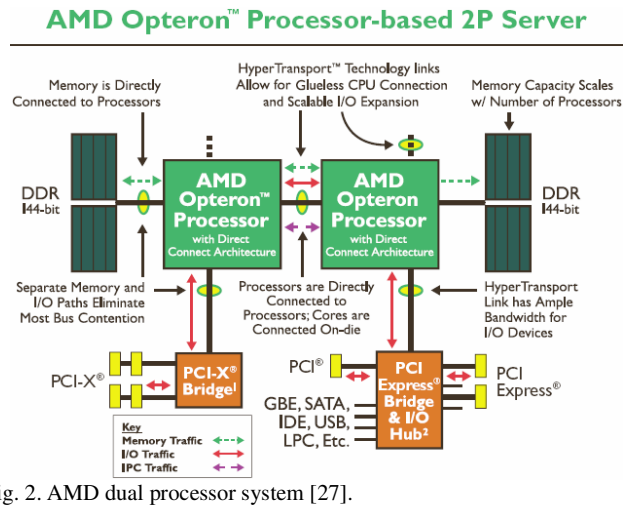


Fig. 2. AMD dual processor system [27].

In Fig.2, the AMD Uniform Memory Access (UMA) multiprocessor system is shown. In this system, each processor has its own local memory, and the processors are connected to each other with a HyperTransport bus. This bus allows processors to access other processor's local memory. Even though the memory is separated, the programmer only sees one continuous block. However, the farther the memory is away from the current processor, the slower it is. The programmer can easily allocate memory to insure that each thread or process is running in the local memory of its current CPU. Cache coherency between the memory and the different processor cache's must also be enforced over the HyperTransport bus.

### B. High Level Solver Designs

When designing a parallel SAT solver, there are many aspects that make it inherently more complex than a single threaded solver. First, all solver threads must be controlled in some manner such that a SAT problem can be loaded and divided dynamically amongst all the solver threads. This control system must also be able to start and then terminate the threads. Secondly, conflict analysis with conflict clause recording is a powerful part of any SAT solver, and in order to take full advantage of this in a parallel solver, some mechanism must allow for the exchange of conflict clauses between solver threads. Furthermore, the solver must still maintain good single threaded performance with these parts included, otherwise the speedup achieved through the use of multiple processors might be negated. For example, PaSAT achieved an excellent speedup of 40 on a benchmark set called longmult using 24 processors [18]. However, the single threaded solver took thousands of seconds to solve each instance in the benchmark set. Good single threaded solvers such as MiraXT and SatELite [7] can solve all 8 instances in this benchmark set in a few hundred seconds on our AMD system, making them faster than PaSAT with 24 processors, although a direct comparison cannot be made as PaSAT was run on older hardware. Lastly, as a side note, MiraXT also achieves super linear speedup on these benchmarks in threaded mode (as do most parallel solver we have seen).

There are many different ways of implementing a parallel SAT solver to realize the points mentioned above, each having its respective advantages and disadvantages. To compare designs, we are going to focus on how different solvers implement clause sharing. This is because clause sharing can significantly improve performance and generally makes up the vast majority of the communication between threads. Here, we will discuss the three main ways we have seen that allow the solver's threads to communicate.

The first and most common way is by using Message Passing. A library such as MPICH [13] is normally used. This method allows for best scaling, allowing for solver threads to be located anywhere as long as they are all connected to some sort of network. This setup was used in GridSAT [19], and in our previous work [23], with both papers showing that speedup can be achieved. However, due to the overhead associated with sending messages, and the limited network

bandwidth, only short clauses are sent (e.g. of length 3 or less in GridSAT), and they are sent in bundles introducing more latency into the clause sharing system.

The second method uses a shared memory "Clause Store" to share clauses. In this system, each thread occasionally sends clauses to the clause store while also checking to see if other new clauses have been added by other threads. PaSAT [21] and ySat [15] both use this design. One difference between the two, is that ySat shared physical copies of each clause, while in PaSAT each thread made its own physical copy of each clause. While the clause store is not as scalable as message passing, the use of shared memory allows longer clauses to be shared (e.g. PaSAT achieved the best speedup sharing clauses with 5 to 10 literals). This system also reduces the latency within the clause sharing system. Note, PaSAT later combined Message Passing and the "Clause Store" in [18] to allow better system scaling.

The third way is the shared memory clause database design that MiraXT uses. Here, the database contains only one physical copy of each clause that threads share. All conflict clauses are added to the database, and each thread selects which clauses it wants to use. This is the reverse of the clause store, in which each thread chooses which clauses it wants to offer or send to the other threads. MiraXT's design allows each thread to consider its current decision stack and status when selecting which conflict clauses it wishes to use. The thread can now decide to add very long clauses that will force implications or cause conflicts, while ignoring short clauses that are already solved by the thread's current variable assignment. This design takes full advantage of the low latency and bandwidth a shared memory database provides. PaSAT did something similar in [18] with mobile agents that contain thread specific information. This information however, was incomplete and not always up-to-date.

## IV. The MiraXT Solver

MiraXT is a zChaff class solver based on MIRA [10,11] but significantly enhanced and modified to allow it to run with multiple threads. MiraXT contains the original MIRA's Early Conflict Dection BCP (ECDB) and Implication Queue Sorting (IQS). In MiraXT, a modified VSIDS algorithm is used, in which all scores over 512 are concatenated so that a bucket sort can be used to sort the list in O(n) time. This allows us to sort the list more frequently keeping it up-to-date, and makes the decision heuristic less greedy. Lastly, it was implemented in C++ using POSIX threads.

### *A. Shared Clause Database*

As mentioned above, MiraXT has one master 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 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 (two simple operations) 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 master 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. Fig.3 shows a top level diagram with threads inserting pointers to clauses into the master clause database. In Fig. 3, $C_x$ represents a pointer to a conflict clause.

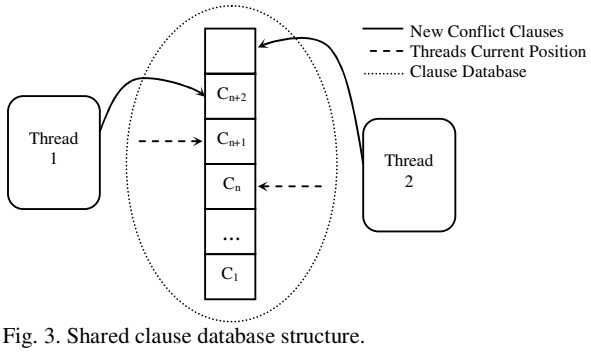


Fig. 3. Shared clause database structure.

Clause deletion is also an important issue. In MiraXT, each thread deletes clauses using an algorithm similar to Berkmin [2] in which older inactive clauses are easier to delete. To facilitate clause deletion efficiently on a multiprocessor system, each thread has one Boolean variable associated with it for every clause. Each clause consists of an array of literals with the first few spots in the array being reserved. These reserved spots specify the clause length, and its unique master database reference number. When a thread deletes its references to a clause, it must set its Boolean variable for that clause using the clause's 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 master database to see if a master delete 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 a master delete is required. If the master 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 master clause deletion procedure at the same time. When the master clauses are deleted, spaces are left in the array the master database uses to keep track of all the clauses. When there are too many open spaces, the array must be compacted to save memory. During this compaction the master clause database is shutdown so that no clauses can be added or retrieved. However, this procedure is done as a quick array copy, and is only very rarely called.

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. This seemed to be one of the problems the solver in [15] suffered, as its authors report that with 4 threads, an average of 10% of the time was spent waiting for locks. This number should be only fractions of a percent, as is indeed the case for MiraXT on most problems. This means that in MiraXT, the majority of the time (over 99% on average) is spent actually solving the problem. This is also good indicator of scalability wrt. future CPU's that contain multiple cores.

### *B. Important Supporting Data Structures*

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 Literal Reference List (WLRL). For each clause, this structure contains two watched literals, and a third literal called a cache variable. The cache variable (CV), is assigned by using the previous watched literal the BCP procedure replaced when it examined the clause. The WRL basically allows each thread in MiraXT to have a condensed reference or copy of every clause. This is done because on the AMD system, when a thread creates a new conflict clause, that clause is located in that CPU's local memory. If other threads want to access it, they must copy it from that thread's local memory into their cache. Reading clauses across the HyperTransport bus can slow the solver down. To combat this problem, the WLRL lists are stored in each threads local memory. In testing on a selection of BMC problems, 84% of clauses with 3 literals or more 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 and the entire clause can be stored here. In any case, this allows MiraXT to better utilize each CPUs cache and local memory. Lastly, this is similar to the work in [10,11,12], however, in these papers, the clauses were directly used.

### *C. Preprocessing*

In this paper, the SatELite solver [7] was used to preprocess all benchmarks. Preprocessors like SatELite or NiVER [26], can greatly reduce the number of variables and clauses in the problem. In addition to this, MiraXT runs a Boolean unit propagation look ahead procedure on all free variables before starting the actual solver. This procedure can eliminate variables by observing that some variables can force the same implications, irrelevant of whether the variable is set to 1 or 0. This is discussed in detail in [14], and used in SAT Solvers like Oepir [9]. This technique is also used in MiraXT during the SAT solver phase when each thread receives a new subproblem, or when the solver has assigned a large number of variables to decision level 0. In these situations, the procedure will only look at variables which could be directly affected by the new variable assignments such as free variables that are in unsolved clauses which contain decision level 0 variables.

These preprocessor techniques eliminate many bad splitting variables (i.e. variables that have no real effect when dividing the problem space, for example, variables that only appear in clauses that are already solved). These variables, if used to divide the problem, will not force new implications, solve new clauses, cause conflicts, or really change the state of the solver in a meaningful way, and in essence, solver threads will end up redundantly searching the same part of the problem space. Also, preprocessing normally improves the solver's single threaded performance.

### *D. Multithreaded Solver Control*

MiraXT contains no controlling master process unlike most 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. Solver threads poll the MCO occasionally to see if there are any messages, or idle threads waiting for a new subproblem.

In principle, MiraXT's threads and the MCO function as follows. Thread 0 starts the solving process on the decision stack given to it after the preprocessing is complete. All other threads start by requesting a subproblem from the MCO and are now waiting to be signaled. Idle threads are not wasting CPU cycles polling, they are put to sleep and awakened using the POSIX *cond_wait* / *cond_signal* commands. Periodically, running threads ask the MCO for any new global events (e.g. problem solved, waiting threads, or time limit). This is done without a lock, and with a simple Boolean variable. If something has happened, a more complicated procedure with a lock will be run. In our example, when thread 0 checks the MCO, it will realize that other threads are waiting for a subproblem. It will then ask the MCO for the decision stack queue lock. This queue contains decision stacks that need to be searched. Once the thread has acquired this lock, it will split its decision stack at decision level 1, and add a decision stack to the queue. It will then signal a sleeping thread, release the decision queue lock, and then continue solving its part of the problem. If there are more threads waiting, they will be randomly served by running threads. No heuristic is used to decide which thread should split its decision stack. If a thread proves its subproblem is unsatisfiable, it will request a new subproblem. If all the threads are waiting for a new decision stack, the problem is unsatisfiable.

### *E. Multithreaded Conflict-Driven Learning*

The conflict analysis procedure in MiraXT is based on the first Unique Implication Point [3]. However, a separate clause addition procedure was added. In MiraXT, the conflict analysis procedure will add a clause pointer to the master clause database. Then the clause addition procedure will be run, asking the master clause database for all new clauses; this includes clauses generated by other threads and its 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 clauses (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. Note, sometimes the thread might decide not to add the conflict clause it just generated because clauses generated by other threads were better, allowing the solver to backtrack further.

The shared memory database MiraXT uses, allows each thread to easily look at all conflict clauses in an efficient manner. Unlike other parallel solvers, MiraXT can be more generous when selecting which clauses to share as there is no real performance penalty associated with sharing. In other parallel solvers, threads are limited to databases that contain only their conflict clauses. So, in other solvers, each thread (or master thread) decides which clauses to distribute, using some simple criteria such as clause length. However, these strategies have a serious drawback in that each thread's current state is not taken into account when sharing clauses. This means useful clauses might not be sent (e.g. because the master process thinks they are too long), and/or useless clauses are sent (e.g. because the threads current decision stack solves the clause). Also, other solvers share clauses by sending them in bundles as messages, or occasionally checking a clause store. Both these designs introduce latency in the knowledge sharing scheme meaning that sometimes clauses are not immediately available where they are needed. In our scheme, these problems are avoided. The shared memory database is what differentiates MiraXT from all other parallel SAT solvers that we know of.

## V. Results and Performance

The results on the IBM BMC 2004 [30] and Industrial 2005 [29] benchmarks are shown in Table 1 and Fig.4. This mix of over 1200 benchmarks contains both SAT and UNSAT instances. The Industrial 2005 benchmarks contain all the grieu05, maris05, narain05, and velev05 sets. The AMD Opteron machine used in this benchmarking section was running a Linux SMP enabled kernel (kernel 2.6.*), contained two Opteron 252 (@2.6 GHz) processors, and had 4 GB of main memory (2 GB of local memory per CPU). The benchmarks were all preprocessed with SatELite first, and then each solver was given 1800 seconds per benchmark. To remove the preprocessing time SatELite required and insure a fair comparison, SatELite was restarted on the preprocessed benchmark with a time limit of 1800 seconds. zChaff version 2004.11.15 and SatELite version 1.0 were used. In Table 1, $T^1$ is the total time used by the solver in thousands of seconds. $T^2$ is $T^1$ minus the time for all the benchmarks that no solver solved, and #S is the number of benchmarks solved. Mira1T and ySat1T are running with 1 thread. Mira2T and ySat2T are running with two threads. The 'a' and 'b' times for MiraXT are different runs of the same solver, included to show the variation in multiprocessor solving times.

TABLE I
Comparison of Solvers

| Solver | IBM BMC 2004 | | | Industrial 2005 | | |
|---|---|---|---|---|---|---|
| | $T^1$ | $T^2$ | #S | $T^1$ | $T^2$ | #S |
| Mira2Ta | 279.4 | 81.4 | 923 | 67.0 | 18.4 | 183 |
| Mira2Tb | 284.5 | 86.5 | 923 | 69.3 | 20.7 | 182 |
| Mira1T | 318.2 | 120.2 | 900 | 75.0 | 26.4 | 178 |
| SatELite | 314.9 | 116.9 | 901 | 77.1 | 28.5 | 176 |
| zChaff | 525.4 | 327.4 | 784 | 84.5 | 35.9 | 175 |
| ySat2T | 707.6 | 509.6 | 709 | 148.0 | 99.4 | 136 |
| ySat1T | 813.0 | 615.0 | 649 | 148.7 | 100.1 | 135 |

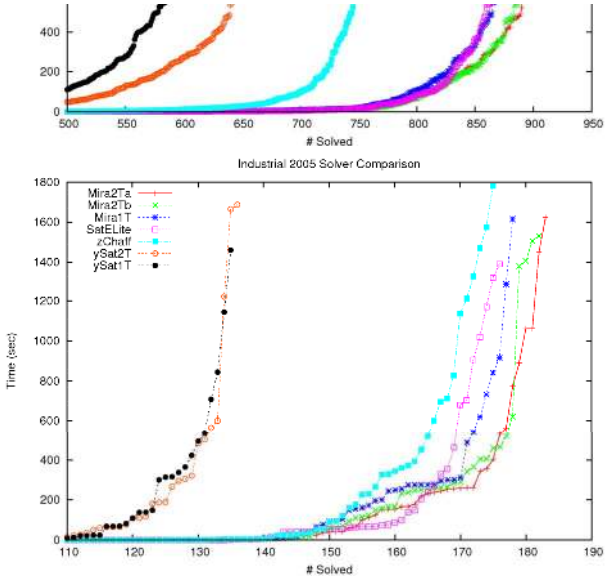


Fig. 4. BMC/Industrial problems solved vs time required.

From Table 1, Mira1T and SatELite are significantly faster than the other solvers on the IBM benchmarks, and all solvers excluding ySat are competitive on the industrial benchmarks. However, Mira2T is significantly faster than the other solvers. ySat is considerably slower then the other solvers on both benchmarks, even when using two processors. Using time $T^2$, MiraXT had an average speedup of 1.43 on the IBM, and 1.36 on the industrial Benchmarks with 2 threads. Focusing only on problems solved by MiraXT (excluding the other solvers results), the speedup is even more pronounced at 1.45 for IBM, and 1.44 for industrial. Speedup was also attained for both SAT and UNSAT instances. On the IBM benchmarks, average speedups of 1.55 and 1.41 were attained for the SAT and UNSAT instances respectively. Remember, these are general benchmarks and not just a select few like the longmult example discussed in Section III.

Next, as can be seen in Fig.4, the curves for Mira1T and SatELite are similar, and zChaff's and ySat's stumbling performance on the IBM benchmarks is quite clear. MiraXT's performance advantage when running with two threads is easy to see. Also, it's interesting that while ySat2T achieved speedup over the single threaded version on the BMC benchmarks, no speedup was attended on the industrial benchmarks, unlike MiraXT.

## VI. Future Perspectives

Preliminary work has also been done on a dual CPU Intel Pentium 4 XEON Machine. It uses a shared memory bus architecture in which both CPU's must share one memory bus. On this system, MiraXT also scales well from one to two threads, providing a performance increase that is just slightly less then the AMD system. This is most likely due to increased memory bus contention. The Intel system however, should be more indicative of dual core CPU performance as dual core CPU's will also share one memory bus. With this in mind, we believe that dual core CPU's should scale almost as well as the AMD system presented here. As for further scaling beyond 2 processors, we foresee no issues. Based on experimental results with 2 processors, MiraXT did not suffer from any lock contention issues, and the amount of work done by the solver (e.g. the number of clauses examined per second) scales almost perfectly. Lastly, roadmaps from both Intel and AMD seem to show future CPU's with significantly larger caches, and faster memory buses. Both should improve MiraXT's multithreaded performance.

## VII. Conclusion

As was shown in this paper, a modern SAT solver can be parallelized using threads to achieve speedup. Significant speedup of 44-45% on both SAT and UNSAT benchmarks was shown when two processors were used. Implementation details that allow the MiraXT solver threads to efficiently work together were discussed, including features that increase single threaded performance. Threaded solvers will likely be the way of the future as Intel, AMD, IBM, and SUN, have all introduced CPUs that contain multiple cores. Utilizing the extra power of these CPUs is and will continue to be a major area of interest in computer science. SAT solvers will have to adapt and become threaded in order to compete with other forms of formal verification. This paper and the ideas presented should provide a good starting point for future research in this area.

## References


[1] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver", *Proceedings of the 38th DAC*, July 2001.

[2] E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust Sat-Solver", *DATE*, 2002.

[3] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver", *ICCAD*, 2001.

[4] L. Zhang, and S. Malik, "Cache Performance of SAT Solvers: A Case Study for Efficient Implementation of Algorithms", *SAT*, 2003.

[5] J. P. Marques-Silva,K. A. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability", *IEEE Transactions on Computers, Vol. 48, pp. 506-521,* 1999.

[6] N. Eén, and N. Sörensson, "An Extensible SAT-Solver", *SAT*, 2003.

[7] N. Eén, A. Biere, "Effective Preprocessing in SAT through Variable and Clause Elimination", *SAT*, 2005.

[8] R. Lawrence, "Efficient Algorithms for Clause-Learning SAT Solvers", *Simon Fraser University Master's Thesis*, 2004.

[9] J. Alfredsson, "The SAT Solver Oepir", *SAT Competition: Solver Descriptions*, 2004.

[10] M. Lewis, T. Schubert, and B. Becker, "Early Conflict Detection Based BCP for SAT Solving", *SAT*, 2004.

[11] M. Lewis, T. Schubert, and B. Becker, "Speedup Techniques Utilized in Modern SAT Solvers - An Analysis in the MIRA Environment", *SAT*, 2005.

[12] I. Lynce, J. and Marques-Silva, "Efficient Data Structures for Fast SAT Solvers", *Technical Report*, 2001.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard", *Parallel Computing*, 1996.

[14] D. Le Berre, "Exploiting the Real Power of Unit Propagation Lookahead", *SAT*, 2001.

[15] Y. Feldman, N. Dershowitz, and Z. Hanna, "Parallel Multithreaded Satisfiability Solver: Design and Implementation", *PDMC*, 2004.

[16] M. Böhm, and E. Speckenmeyer, "A Fast Parallel SAT-Solver - Efficient Workload Balancing", *Annals of Mathematics and Artificial Intelligence*, 1996.

[17] W. Blochinger, C. Sinz, W. Küchlin, "A Universal Parallel SAT Checking Kernel", *PDPTA*, 2003.

[18] W. Blochinger, C. Sinz, and W. Küchlin, "Parallel Propositional Satisfiability Checking with Distributed Dynamic Learning", *Parallel Computing*, 2003.

[19] W. Chrabakh, and R. Wolski, "GridSAT: A Chaff-based Distributed SAT Solver for the Grid", *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2003.

[20] B. Jurkowiak, Chu Min Li, and G. Utard, "Parallelizing Satz Using Dynamic Workload Balancing", *SAT*, 2001.

[21] C. Sinz, W. Blochinger, W. Küchlin, "PaSAT - Parallel SAT-Checking with Lemma Exchange: Implementation and Applications", *SAT*, 2001.

[22] W. Blochinger, C. Sinz, and W. Küchlin, "Distributed Parallel SAT Checking with Dynamic Learning using DOTS", *PDCS*, 2001.

[23] T. Schubert, M. Lewis, B. Becker, "PaMira - a Parallel SAT Solver with Knowledge Sharing", *International Workshop on Microprocessor Test and Verification*, 2005.

[24] H. Zhang, M. Bonacina, and J. Hsiang, "PSATO: A Distributed Propositional Prover and its Application to Quasigroup Problems", *Journal of Symbolic Computation*, 1996.

[25] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving", *Communications of the ACM, vol. 5, pp 394-397*, 1962.

[26] S. Subbarayan, D. Pradhan, "NiVER: Non Increasing Variable Elimination Resolution for Preprocessing SAT instances.", *SAT*, 2004.

[27] http://www.amd.com/us-en/assets/content_type/DownloadableAssets/ PID30291H_2P_server_competitive_comp.pdf

[28] G. Moore, "Cramming More Components Onto Integrated Circuits", *Electronics*, 1965.

[29] SAT2005 benchmarks, SATLIB: http://www.satlib.org

[30] IBM BMC Benchmarks: http://www.haifa.il.ibm.com /projects/verification/RB_Homepage/bmcbenchmarks.html