Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 128 (2005) 75–90
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Parallel Multithreaded Satisfiability Solver: Design and Implementation

Yulik Feldman[a], Nachum Dershowitz[b] and Ziyad Hanna[a]

[a] *Intel Corporation, Haifa,* {yulik.feldman, ziyad.hanna}@intel.com

[b] *School of Computer Science, Tel Aviv University, Tel Aviv, Israel,* nachum.dershowitz@cs.tau.ac.il

Abstract

We describe the design and implementation of a highly optimized, multithreaded algorithm for the propositional satisfiability problem. The algorithm is based on the Davis-Putnam-Logemann-Loveland sequential algorithm, but includes many of the optimization techniques introduced in recent years. We provide experimental results for the execution of the parallel algorithm on a variety of multiprocessor machines with shared memory architecture. In particular, the detrimental effect of parallel execution on the performance of processor cache is studied.

Keywords: satisfiability parallel multithreaded DPLL cache performance

1 Introduction

This paper describes the design and implementation of a highly optimized, parallel multithreaded algorithm for solving the propositional satisfiability problem (SAT). SAT is a fundamental problem in the theory of computation, one that has been studied extensively for more than four decades, ever since the introduction of the first algorithm for its solution in 1960 [5]. Eleven years later it became the first problem proven to be NP-complete, in a famous paper by Cook [3]. Nowadays, the problem evidences great practical importance in a wide range of disciplines, including hardware verification [17], artificial intelligence [10], computer vision [2] and others. Indeed, one survey of satisfiability [6] contains over 200 references to applications.

In spite of its computational complexity, there is strong demand for high-performance SAT-solving algorithms in industry. Over the years, many dif-

1571-0661
doi:10.1016/j.entcs.2004.10.020

ferent approaches and optimizations have been developed to tackle the problem more efficiently. Algorithms have evolved gradually by extending existing methods with new, more powerful, optimizations. Current research on propositional satisfiability is focused on two classes of SAT solving methods: complete algorithms and incomplete procedures. Complete methods, mostly represented by *backtrack search* algorithms, identify both satisfiable and unsatisfiable problems, and show reasonably good performance on both types. Incomplete methods, mostly represented by variations of *local search* procedures, perform much faster on satisfiable problems, but incur the price of not always being able to demonstrate unsatisfiability. The fact that backtrack search algorithms are able to cope with unsatisfiable problems usually makes them the preferred choice in domains where proofs of unsatisfiability are required.

Our implementation of a complete backtrack-search parallel algorithm incorporates most of the state-of-the-art sequential technologies introduced in recent years. The emphasis has been on providing an efficient portable implementation that would work on any typical multiprocessor workstation in an industrial environment and improve runtime, as compared with the core sequential algorithms, by distributing the workload over several processors on the machine. To make sure the implementation is efficient and that low-level implementation details are properly understood, the implementation of the solver was not based on existing publicly available source code of other solvers, but instead was designed and coded from scratch. The implementation was carefully crafted to enable a direct comparison of its behavior with existing sequential algorithms. After the implementation of the algorithm was completed, its behavior in different real-life environments was measured. In particular, the effect of concurrent execution of multiple threads on the behavior of platform architecture primitives, such as processor cache, bus utilization and resource allocation, was investigated. The main contribution of this work is that it shows the general disadvantageousness of parallel execution of a backtrack-search algorithm on a multiprocessor workstation, due to increased cache misses.

The remainder of this paper is organized as follows: Section 2 briefly describes the SAT problem. Section 3 gives an overview of the state-of-the-art techniques used in the implementation of sequential solvers, while Section 4 describes the methods used to parallelize the core sequential algorithms. Section 5 reports on the results of experimental runs of the implemented parallel SAT solver. Section 6 describes related work in the area of parallel SAT solving, and is followed by a brief concluding section.

2 The SAT Problem

The propositional satisfiability problem can be formulated easily in one sentence: given a boolean formula, check whether an assignment of boolean truth values to the propositional variables in the formula exists, such that the formula evaluates to true. If such an assignment exists, the formula is said to be *satisfiable*; if no such assignment exists, it is *unsatisfiable*. For a formula with n variables, there are 2^n possible truth assignments.

Although there are many ways to represent a boolean formula, *conjunctive normal form* (CNF) is most frequently used for this purpose. In general, it is not a limitation to use CNF for the representation of formulas, since any boolean formula can be transformed to an equivalently satisfiable formula in CNF (with extra variables) in polynomial time [14]. In CNF, the variables of the formula appear in *literals*, which are either a lone variable (x) or the negation of a variable ($\bar{x}$). Literals are grouped into *clauses*, which represent a disjunction (logical or) of the literals they contain. A single literal can appear in any number of clauses. The conjunction (logical and) of all clauses represents a formula. For example, the CNF formula $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_3) \wedge (x_1 \vee x_3)$ contains three clauses: $x_1 \vee \bar{x}_2$, $\bar{x}_3$ and $x_1 \vee x_3$. Two literals in these clauses are positive (x_1, x_3) and two are negative ($\bar{x}_2$, $\bar{x}_3$). Note that for a variable assignment to satisfy a CNF formula, it must satisfy each of its clauses. For example, if x_1 is true and x_3 is false, then all three clauses are satisfied, regardless of the value of x_2.

3 Sequential SAT

The sequential SAT solving algorithm employed in our parallel implementation is based on the commonly used Davis-Putnam-Logemann-Loveland algorithm (DPLL) [4], which performs a methodical enumeration of assignments, looking for one that satisfies the formula. To build the assignments, the algorithm chooses variables in a certain order and incrementally assigns a value to each. As long as the resulting *partial* assignment does not falsify the formula, it continues to choose variables and assign values. If the resulting partial assignment falsifies the formula, the algorithm assigns the opposite value to the last variable chosen and checks whether the resulting assignment still falsifies the formula. If not, it proceeds with assigning values to more variables. If the formula is falsified regardless of the value of the most recently assigned variable, it backtracks to the previously chosen variable and assigns it the opposite value. The algorithm continues the search in a similar fashion, assigning, reassigning and unassigning values until all variables are assigned and the formula is satisfied, or, alternatively, until all possible assignments

are accounted for, without finding a satisfying assignment. Note that DPLL does not always explicitly visit all 2^n assignments, since it backtracks once an unsatisfying partial assignment is found. Of course, the complexity of the algorithm remains exponential.

The original DPLL algorithm incorporated an optimization technique called *unit propagation*, or *boolean constraint propagation* (*BCP*). If the current partial variable assignment causes all but one literal of some clause to have the value false, then the remaining literal must be assigned true in order not to falsify the clause and, consequently, the whole formula. Such a literal is called a *unit literal*, and such a clause is a *unit clause*. When a unit clause is found, the algorithm chooses the variable of a unit literal next, and assigns it the value that makes the clause true.

In recent years, many sophisticated optimization techniques have been introduced to improve the performance of the core DPLL algorithm. These include *watched literals*, *conflict analysis*, *non-chronological backtracking*, *variable state independent decaying sum (VSIDS) decision heuristics* and *restarts*. Refer to [11,13] for a description of these techniques, all of which have been incorporated in the parallel solver presented here.

The actual performance of our solver in a single-threaded configuration on a benchmark suite of about 500 industrial tests is comparable to, although slightly below, the performance of the award-winning zChaff sequential solver [13]. The total runtime of our solver on this benchmark suite was about 10% higher than that of zChaff, and its total memory consumption was about 12% smaller. Our solver was able to solve seven more tests, within the time and memory limits set during the experiment.

4 Parallel SAT

To parallelize the sequential DPLL algorithm, the search space is partitioned into several disjoint parts that are treated in parallel. An important characteristic of the SAT search space is that it is hard to predict the time needed to complete a specific branch of the search space. Consequently, it is impractical to partition the search space statically at the beginning of the algorithm, since an incorrect prediction of the complexity of the chosen partitions would result in an uneven workload distribution, and, concomitantly, in reduced efficiency of the algorithm. To cope with this problem, the implemented parallel algorithm dynamically partitions the search space, assigning available work to the available threads during run-time.

4.1 Search space partitioning

To partition the search space, the algorithm uses the concept of *guiding path*, first introduced in [1]. The guiding path describes the current state of the search process. It does so by recording the list of variables to which the algorithm assigned a value up until the given point of execution. For each recorded variable, the guiding path associates the currently assigned boolean value, as well as a boolean flag that says whether there has been an attempt to assign both boolean values to the given variable or whether the currently assigned value is the only one for which the assignment has been attempted. A variable for which there was an attempt to assign both boolean values is said to be *closed*, while one for which there was an attempt to assign only one value is *open*. Open variables represent junctions on the guiding path that lead to yet unexplored search space. In the sequential SAT-solving algorithm, which can be seen as a special case of a parallel SAT-solving algorithm working with a single thread, the guiding path represents the internal stack of the partial assignments. Variables that are assigned new values are pushed onto the stack, and, therefore, are added to the end of the current guiding path. Variables that are removed from the stack as a result of backtracking are removed from the end of the guiding path.

Since a single thread explores only the currently assigned value of the open variables, the search algorithm may be parallelized by letting other threads explore the search space defined by the open variables on the guiding path of the current thread. The other threads start their execution by assigning the variables that precede the selected open variable on the guiding path, with the values stored on the guiding path, and by flipping the value assigned to the open variable. The selected open variable is then marked "closed" to prevent other threads from following the paths already being explored. Note that each running thread maintains a private guiding path associated with its execution state. The available threads are then free to select any open variable from any existing guiding path to pick up a new task. The thread that has the selected open variable, and the thread that selects that variable, are said to be in a parent-child relationship: the thread with the selected open variable is the *parent*; the one that selects the variable is the *child*. Running threads form a conceptual tree, wherein nodes represent threads and edges represent parent-child relationships.

4.2 Task scheduling

The execution of the parallel algorithm starts with a thread tree consisting of a single node that is assigned the task of solving the whole problem instance.

As execution of the first thread begins, a number of open variables appear on the guiding path of the first thread. A second thread picks one of the open variables and joins the thread tree as a child of the root and explores a subspace of the solution space being explored by the root. The other threads choose an open variable from one of the running threads and join the tree, exploring subspaces of those that are being explored by their parent threads. The tree grows as long as available threads join the execution of the algorithm.

A working thread finishes the execution of its current task in one of two fashions: either the thread finds an assignment to all variables such that the whole formula is satisfied, or the thread figures out that no such assignment exists in its subspace. In the first case, the parallel algorithm is stopped and the solution is reported. In the second case, the outcome does not necessarily mean that the whole formula is unsatisfiable, since the subspaces being explored by other threads were not searched by the current thread. In this case, the completed thread picks up an open variable from one of the other threads and starts exploration of the corresponding subspace. The thread that finished execution of its latest task is removed from the tree and joins it again at a different branch. If no available open variables exist at the time an available thread looks for a new task, the thread is temporarily suspended until an open variable appears. If all threads finish their execution and are in suspension while waiting for an available task, this indicates that the search space has been fully explored and the problem is unsatisfiable.

Note that with such dynamic partitioning of the search space, the work load is evenly distributed between the working threads, and these threads are all kept busy until the problem is solved. To minimize the thread waiting times and the time needed to find a new task, a list of available tasks is maintained. When a new open variable is introduced by a thread, the thread adds a description of a task that is associated with the new variable to the list of available tasks. Since the number of open variables is usually much larger than the number of working threads, the threads add new tasks to the list only until a certain threshold on the size of the list is reached. When a thread completes the execution of the current task, it chooses an available task from the list and removes the task from the list. The other threads promptly add a new task to the list, maintaining its size around the threshold. Due to the large discrepancy between the number of open variables and the number of threads, thread-waiting times on an empty list are very short. These times are restricted to the stage just after the beginning of execution of the algorithm, when the list is still empty, up to the time just prior to the completion of execution, when no more open variables remain.

To reduce the overhead of reinitializing the state of the threads when they

switch from execution of one task to another, the available tasks are chosen in such a way that the expected running time of each individual task is higher. This is achieved by choosing open variables that are closest to the beginning of the guiding paths. In each thread, such a single open variable is chosen as a candidate for entering into the list of available tasks. The list of available tasks is sorted by the same means, according to the length of the guiding path leading to the open variables in the tasks. This approach prevents frequent task switches that would create additional overhead compared to the sequential algorithm. Aside from reducing the overhead of task switches, the choice of these variables also eliminates the need to implement a complex messaging mechanism between threads, as explained next.

4.3 Conflict clause exchange

Although the threads work quite independently of each other, the parallel algorithm requires special treatment of the *conflict clauses* produced by each thread. Conflict clauses are clauses generated by the conflict analysis algorithm, described in [11]. The clauses, similar to some other internal data structures used by the parallel algorithm, have thread-specific data associated with them. This data should be initialized in the context of each thread to let the threads benefit from the existence of conflict clauses. Since the clauses are generated by specific threads, the information about the generated clauses should be distributed to the other threads. This is done by maintaining a list of generated conflict clauses that is accessible from all threads. When a conflict clause is generated by a thread, the thread that generated it puts it onto the list. The others threads frequently check the existence of new clauses on this list. If a thread detects that a new clause that is not yet initialized in the context of that thread has been added to the list, it initializes the clause in its own context, and marks the clause as initialized in its context. Whenever a clause is initialized in the contexts of all working threads, it is removed from the list.

The distribution of conflict clauses between threads improves the effectiveness of the search performed by each thread, much like it does in the sequential SAT solving algorithm. It also eliminates the need to implement a complex messaging mechanism between threads, which would allow the threads to terminate the execution of tasks in other threads when they discover that these tasks are not essential for solving the problem. The need for such termination signals arises when a thread finds a conflict and backtracks over several variables to resolve the conflict. If one of the backtracked variables became closed as a result of another thread starting exploration of the corresponding subspace of the solution space, this exploring thread should be informed

that its current task is superfluous, since it leads to a conflict found by the current thread. The implementation of the termination signals may be complex and inefficient due to the need to implement synchronization mechanisms protecting the thread data from simultaneous access from different threads. Fortunately, if the distribution of conflict clauses is implemented, it becomes unnecessary to signal other threads explicitly. When the current thread finds a conflict, aside from backtracking over several variable assignments, it also generates a conflict clause that describes the reason for the conflict, and puts the clause on the list of conflicts. Once the other thread detects the presence of this clause on the list and initializes it in its own context, it will be forced to backtrack itself to avoid the conflict described in the conflict clause. Since the tasks are created based on the open variable found closest to the beginning of the corresponding guiding path, it is guaranteed that no more open variables are on the guiding path of the thread, and the backtrack algorithm will terminate execution of the task once it reaches the beginning of the guiding path.

4.4 Thread synchronization overhead

From an implementation point of view, the list of available tasks and the list of conflict clauses not yet learned by all threads are the only two data structures that implement synchronization mechanisms to protect data from simultaneous access from different threads. Both new task generation and new conflict clause generation are relatively infrequent tasks compared to the rest of the work being performed by threads. This makes the synchronization overhead of these data structures insignificant compared to overall performance.

Aside from these two global data structures, the solver maintains another global structure that represents the clauses of the formula being solved. However, since this data structure is read-only, it does not require thread synchronization.

The remaining data structures are accessed by the single thread owning them, and, as such, do not require any thread synchronization. As a result, the introduction of thread synchronization mechanisms does not impose significant overhead on the performance of the core sequential algorithm (see Section 5 for more details). Figure 1 displays the high-level architecture of the solver.

5 Experimental Results

In this section, experimental results of running the implemented parallel multithreaded SAT solver on a variety of machine configurations are provided. It should be noted that, in general, it is difficult to precisely measure the per-

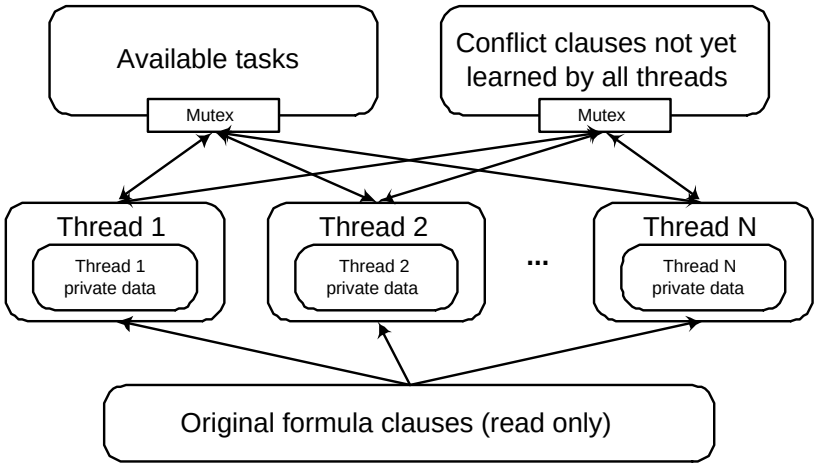

Fig. 1. High-level architecture of the parallel multithreaded satisfiability solver

formance of the solver due to the high variance of run-times of consecutive invocations of the same test in the same environment. This may be attributed to a combination of two factors. The first is the inherent indeterminism of the parallel execution of a multithreaded program. The second factor is the unpredictably unbalanced structure of the search space of the SAT problem. These two factors together make it difficult to measure the performance of an individual test, which may change in order of magnitude from run to run. To minimize the effect of indeterminism on the results, the tests in this section were run several times and the total running times were recorded. The standard deviation was in the range of 20–30%.

On the first set of tests, overall performance of the solver on a single medium-size SAT problem was measured—over a variety of different machine architectures with different numbers of concurrently running threads. The particular SAT problem was chosen in such a way that it is complex enough to objectively test the overall performance of the solver and small enough to allow running multiple tests within a reasonable timeframe. It was also chosen so that memory consumption does not cause a performance bottleneck and processor performance alone is being tested. The problem is `dlx2_aa` from the "Superscalar Suite 1.0" of Velev [18], and represents the correctness criteria for the 2-issue superscalar DLX processor with in-order execution, having 2 pipelines of 5 stages each. The problem, with 490 variables and 2804 clauses, is unsatisfiable. The single-threaded configuration of the solver requires about 7000 decisions and 120,000 implications to conclude that it is unsatisfiable. During the run, the solver consumes about 1.5MB of heap memory.

Table 1 shows the different machine configurations used in the tests. The

Table 1
Machine configurations

	OS	Processor Type	MHz	HT	CPU	L2
A	Windows 2000 AS	Pentium III	700	No	4	1024K
B			500	No	2	512K
C	Windows XP	Mobile Pentium III	800	No	1	512K
D		Pentium M	1700	No	1	1024K
E	Linux RH 7.1	Pentium 4	2400	No	1	512K
F		Xeon	2200	Yes	1(2)	512K
G			2400	Yes	2(4)	512K
H	Linux RH AW 2.1	Itanium 2 (64 bit)	1300	No	2	(*)

HT column specifies whether the given processor has Intel® Hyper-Threading Technology (HT) enabled. When HT is enabled, each physical processor is perceived by the OS as two logical processors, enabling more concurrency between threads in the system (the number of logical processors is shown in parentheses). The L2 column specifies the amount of L2 cache in each processor in kilobytes. All processors have two levels of cache, with the exception of the Itanium 2 processor (Configuration H), which has three levels of cache (256K of L2 cache and 3072K of L3 cache). Table 2 shows the overall performance of the SAT solver on each configuration in Table 1, with a different number of concurrently working threads. Numbers are given in seconds and represent the sum of runtimes for 10 consecutive invocations of the same test. The overhead for executable startup, thread initialization and parsing of the problem was in the range of 0.1-0.2 seconds per invocation. The last column gives the ratio of performance of the configuration with four threads to the single-threaded configuration.

As this set of tests shows, the overall performance of the solver, not only does not improve with the increased number of concurrently working threads, but becomes worse when the number of working threads is increased. Performance degradation is especially severe on systems that have more than one processor, whether physical or logical. The least degradation is observed with Configurations C, D and E, which are single processors. The worst degradation occurs with Configuration G, with two processors and HT enabled. The tendency for performance to degrade with an increased number of threads is not limited to configurations shown in Table 2, with up to four threads; per-

Table 2
Performance of SAT solver with different numbers of working threads

Configuration	One	Two	Three	Four	Four:One
A	13	15	61	89	6.8
B	20	21	42	47	2.4
C	14	16	19	22	1.6
D	13	15	14	15	1.2
E	7	7	7	10	1.4
F	8	20	27	53	6.6
G	6	55	195	168	28.0
H	6	52	86	107	17.8

formance continues to degrade as the number of working threads is increased beyond four. A similar picture was observed when the solver was invoked on other problems.

The above results suggest that there is some kind of interference between threads running on different processors, which causes the performance degradation. To locate possible sources of such interference, a detailed analysis of algorithm performance on a single machine configuration with a varying number of threads was done. The Intel® VTune™ Performance Analyzer [8] was used to collect data and to perform the analysis. The initial investigation of solver process behavior relative to the other processes in the system, load distribution of the threads inside the process and the distribution of function calls inside the threads did not reveal the cause for the degradation of performance as the number of working threads increases. Independent of the number of working threads, the solver process took a large part of the total processor load, the load distribution between process threads was even, and the same function call patterns appeared in the performance bottlenecks inside the threads. Consistent with performance reports of other DPLL satisfiability solvers, for about 70% of total running time the threads were busy running boolean constraint propagation algorithms, while this number did not change with the number of working threads. The only thing that changed with the increased number of threads was the time that different functions spent waiting on synchronization locks for shared data structures. However, even in the case of four running threads, the total waiting time did not reach 10% of the total running time, a percentage that could not explain the performance

degradation observed in the above tests.

With the help of VTune's sampling performance analysis, it was found that the average number of processor clockticks needed to execute a single processor instruction (CPI, clockticks per instruction) grows significantly with the increased number of working threads. While the CPI of the solver process was about 1.4 in the configuration with one thread, which is considered very good for this class of processors, the CPI grew to about 3.7 in the configuration with four threads, which is considered poor.

To investigate the cause of this degradation further, the behavior of processor-monitoring events was analyzed. The processor-monitoring events are hardware-level processor-specific counters that enable monitoring of low-level processor events, such as cache misses and bus utilization. (For a detailed description of processor-monitoring events, refer to [7].) A total of more than 200 different tests have been run and detailed statistics have been collected. Due to lack of space, Table 3 shows only the most interesting results. All tests in the table were run on the same SAT problem on the same machine (Configuration B from Table 1). Instead of showing the raw values of various processor-monitoring events, the table shows the ratios of the events to other related basic events. These ratios make the numbers independent of the actual runtime of a process. In particular, the increased runtimes of tests due to an increased number of working threads have no effect on the ratios. For example, the "Instructions Decoded / Clockticks" ratio represents the average amount of decoded instructions per processor clocktick, independent of how many clockticks have actually been executed.

As Table 3 demonstrates, most of the above ratios are strongly affected by the increased number of threads. The more threads running, the worse the ratios look. The most seriously affected ratios are the increased cache and memory misses which slow down the execution very significantly. The average "L2 M-state Lines Allocated / DMRs" is 0.0005 when one thread is running, while it is more than 0.0053 when two or more threads are running, a tenfold increase (!). It is important to note, once again, that these numbers are independent of actual execution time, which varies with the number of threads.

Several factors may lead to the increased cache misses. One is that the essence of the parallel SAT solving algorithm requires that most of the auxiliary data structures storing the current state of the algorithm are duplicated for each additional thread. The only data that can be shared between threads are the sets of literals of the formula clauses. This does not constitute a major part of the total processed data. The increased number of memory allocations results in an increased number of cache misses. Table 4 shows the amount of

Table 3
Ratios of processor-monitoring events with different number of working threads

Ratio	**One**	**Two**	**Three**	**Four**
Partial Stall Cycles / Clockticks	0.0216	0.0413	0.0483	0.0427
Resource Related Stalls / Clockticks	0.2758	0.7620	0.4565	0.4439
L2 Cache Reads / DMRs (*)	0.0135	0.0175	0.0303	0.0314
L2 Cache Writes / DMRs	0.0017	0.0041	0.0096	0.0088
L2 M-state Lines Allocated / DMRs	0.0005	0.0053	0.0094	0.0066
L2 M-state Lines Evicted / DMRs	0.0004	0.0036	0.0109	0.0082
External Bus Cycles / Clockticks	0.0008	0.0070	0.0079	0.0095
Instructions Decoded / Clockticks	0.7908	0.5984	0.4855	0.4511
L2 Cache Request Misses / DMRs	0.0013	0.0075	0.0126	0.0096

(*) Data Memory References

Table 4
Heap memory allocation with different number of working threads

Decisions	**1000**	**2000**	**3000**	**4000**	**5000**	**6000**	**7000**
One thread	872	968	1036	1104	1308	1464	1596
Two threads	1172	1408	1528	1624	1756	1828	1912
Three threads	1416	1472	1584	1668	1928	1976	2100
Four threads	1648	1768	2080	2232	2340	2392	2592

allocated heap memory as a function of the number of working threads and the number of decisions made by the solver. Note that when more than one thread is used, the actual number of decisions made during the solution of the tested SAT problem may vary from about 2000 to about 9000, due to nondeterminism of the algorithm. The data in Table 4 shows the approximate memory allocation (in kilobytes) made during solver invocations that resulted in a total of about 7000 decisions.

In addition to an increased number of memory allocations, the algorithm is unable to process the data in a linear fashion to allow pre-fetching of coming data. Rather, data is accessed in a nearly random order, inside a single thread and, in addition, with no correlation between different threads. Frequent

accesses to data from an increased number of locations also result in increased cache misses. When the algorithm is run on a multiprocessor machine, the situation is worsened by the fact that a change of data by one processor invalidates the cache lines holding the memory region surrounding the changed data in other processors.

The data structures and algorithms of modern SAT solvers are highly optimized with regard to cache misses, so the sharp increase in the number of cache misses in a multithreaded environment seemingly overweighs the potential advantages of parallel execution of parts of the problem on a single multiprocessor machine.

These hypotheses as to the root causes of the performance degradation are based on the experimental results and on a detailed analysis of the implemented algorithm, after having invested considerable effort in an attempt to optimize cache behavior. Still, it is possible that some alternate organization of data structures or different sequential or parallel algorithms might reduce the number of cache misses within a multithreaded environment.

6 Related Work

Research on parallelizing SAT solving algorithms can be traced back to a 1994 paper by Bohm and Speckenmeyer [1], who presented a parallelization of a simple sequential Davis-Putnam (DP) SAT solving algorithm for k-SAT problems on a parallel MIMD machine consisting of 320 T800 transputers. The authors showed a linear speed-up with increasing numbers of processors.

In subsequent years, a number of works in the field of parallel SAT algorithms were published. These included parallelizing a more advanced version of DP/DPLL algorithms, such as PSATO [19] and parallel Satz [9], parallelizing local search algorithms [12] and hardware-based approaches [20]. One of the most interesting is the implementation of PaSAT [15,16], a parallel version of a DPLL-based SAT solver that incorporates a number of recently introduced techniques, such as conflict analysis, non-chronological backtracking and dynamic learning. The authors make a special emphasis on studying the behavior of dynamic learning in a parallel environment and its effect on overall performance. The parallel solver was run on a cluster of 24 Sun workstations, and variations of different dynamic learning parameters on several test cases were observed. In many cases, the authors achieves linear, and even super-linear, speed up in terms of the number of running threads.

There are two main aspects in which the work presented in the current paper differs from the above works. First, a substantial effort has been made to implement efficiently most published state-of-the-art sequential SAT solving

techniques, making the performance of the single-threaded algorithm directly comparable to other modern SAT solvers. This allowed the studying of the behavior of parallel execution of the algorithm in a real-life environment, where it had to coexist with other implemented optimizations of the SAT solving algorithm. This also made it possible to observe the negative effect on otherwise highly optimized cache performance of the sequential algorithm.

The other major distinction between this and previous works is that we have investigated the parallel execution of a SAT solving algorithm on a single multiprocessor workstation with shared memory architecture, as opposed to executing on a cluster of network-connected machines. In a typical industrial environment, it is usually difficult to dedicate a cluster of network-connected machines to the solution of a SAT problem, due to the lack of sufficient resources. On the other hand, it is quite common for one or more processors on a company workstation to be idle, since the operating system is unable to distribute the workload of a single-threaded SAT solving algorithm to other processors. However, while it is possible to achieve a linear speed-up on a cluster of network-connected machines, the effect of shared memory architecture on cache performance seemingly diminishes the advantages of parallel execution on a single multiprocessor workstation.

7 Conclusion

The previous sections presented experimental results of running a highly optimized parallel SAT solving algorithm on a single multiprocessor workstation with shared memory architecture. The results show a very significant detrimental effect on cache performance, and, consequently, on total run-time. Cache performance is so greatly affected that total run-time grows with the increased number of running threads, in spite of the workload distribution among different processors. This effect remains similar on a variety of hardware and system configurations, with the tendency to become stronger as the number of processors increases.

The structure of the SAT problem, and the backtrack search SAT algorithm, make it very difficult to adjust the data structures or the algorithm for better cache locality during concurrent execution of parts of the problem. As a result, there seems to be no practical advantage in attempting to optimize the backtrack search algorithm by letting it execute concurrently on a multiprocessor workstation.

References

[1] Max Bohm and Ewald Speckenmeyer. "A fast parallel SAT-solver — efficient workload balancing", URL: http://citeseer.ist.psu.edu/51782.html, 1994.

[2] Ronald T. Chin and Charles R. Dyer. *Model-based recognition in robot vision*, ACM Computing Surveys, 67–108, 1986.

[3] Stephen A. Cook. *The complexity of theorem proving procedures*, Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing, 151–158, 1971.

[4] Martin Davis, George Logemann and Donald W. Loveland. *A machine program for theorem proving*, Journal of the ACM, 394–397, 1962.

[5] Martin Davis and Hilary Putnam. *A computing procedure for quantification theory*, Journal of the ACM, 201–215, 1960.

[6] Jun Gu, Paul W. Purdom, John Franco and Benjamin W. Wah. "Algorithms for the satisfiability (SAT) problem: A survey", URL: http://citeseer.ist.psu.edu/56722.html, 1996.

[7] Intel Corp. "IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture", URL: http://developer.intel.com/design/Pentium4/ documentation.htm, 2003.

[8] Intel Corp. "Intel® VTune™ Performance Analyzer", URL: http://www.intel.com/software/products/vtune/vpa/index.htm, 2004.

[9] Bernard Jurkowiak, Chu Min Li and Gil Utard. *Parallelizing Satz using dynamic workload balancing*, Electronic Notes in Discrete Mathematics, **9** (2001).

[10] Henry Kautz and Bart Selman. *Unifying SAT-based and graph-based planning*, Workshop on Logic-Based Artificial Intelligence, 1999.

[11] João P. Marques-Silva and K. A. Sakallah. *Conflict analysis in search algorithms for propositional satisfiability*, Proceedings of the IEEE International Conference on Tools with Artificial Intelligence, 1996.

[12] Simone L. Martins, Celso C. Ribeiro, Mauricio C. Souza. *A parallel GRASP for the Steiner problem in graphs*, Workshop on Parallel Algorithms for Irregularly Structured Problems, 1998.

[13] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. *Chaff: engineering an efficient SAT solver*, Proceedings of the 38th Design Automation Conference, 2001.

[14] David A. Plaisted and Steven Greenbaum. *A structure-preserving clause form translation*, Journal of Symbolic Computation **2** (1986), 293–304.

[15] Carsten Sinz, Wolfgang Blochinger and Wolfgang Küchlin. *PaSAT - parallel SAT-checking with lemma exchange: implementation and applications*, Proceedings of SAT2001.

[16] Carsten Sinz, Wolfgang Blochinger and Wolfgang Küchlin. *Parallel propositional satisfiability checking with distributed dynamic learning*, Parallel Computing **29(7)** (2003), 969–994.

[17] Miroslav N. Velev and Randal E. Bryant. *Effective use of Boolean satisfiability procedure in the formal verification of superscalar and VLIW microprocessors*, Proceedings of the Design Automation Conference, 226–231, June 2001.

[18] Miroslav N. Velev. "Superscalar Suite 1.0", URL: http://www.ece.cmu.edu/ ∼mvelev, 1999.

[19] Hantao Zhang, Maria Paola Bonacina and Jieh Hsiang. *PSATO: a distributed propositional prover and its application to quasigroup problems*, Journal of Symbolic Computation, 1996.

[20] Ying Zhao, Sharad Malik, Matthew Moskewicz and Conor Madigan. *Accelerating Boolean satisfiability through application specific processing*, ISSS, 2001.