# Asynchronous Multi-Core Incremental SAT Solving

Siert Wieringa and Keijo Heljanko*

Aalto University, School of Science
Department of Information and Computer Science
PO Box 15400, FI-00076 Aalto, Finland
siert.wieringa@aalto.fi keijo.heljanko@aalto.fi

**Abstract.** Solvers for propositional logic formulas, so called SAT solvers, are used in many practical applications. As multi-core and multi-processor hardware has become widely available, parallelizations of such solvers are actively researched. Such research typically ignores the incremental problem specification feature that modern SAT solvers possess. This feature is, however, crucial for many of the real-life applications of SAT solvers. Such applications include formal verification, equivalence checking, and typical artificial intelligence tasks such as scheduling, planning and reasoning.
We have developed a multi-core SAT solver called Tarmo, which provides an interface that is compatible with conventional incremental solvers. It enables substantial performance improvements for many applications, without requiring code modifications. We present the *asynchronous interface*, a natural extension to the conventional solver interface that allows the construction of efficient application specific parallelizations. Through the asynchronous interface multiple problems can be given to the solver simultaneously. This enables conceptually simple but efficient parallelization of the solving process. Moreover, an asynchronous solver is easier to run in parallel with other independent tasks, simplifying the construction of so called coarse grained parallelizations. We provide an extensive experimental evaluation to illustrate the performance of the proposed techniques.

## 1 Introduction

*Propositional satisfiability* (typically abbreviated SAT) is the problem of finding a satisfying truth assignment for a given propositional logic formula, or determining that no such assignment exists. This classifies the formula as respectively *satisfiable* or *unsatisfiable*. SAT is an important theoretical problem as it was the first problem ever to be proven NP-complete [9].

Despite the theoretical hardness of SAT, current state-of-the-art decision procedures for SAT, so called *SAT solvers*, have become surprisingly efficient. Subsequently these solvers have found many industrial applications. Such applications are rarely limited to solving just one decision problem. Instead, a single application will typically solve a series of related problems. Modern SAT solvers handle such problem sequences through their *incremental SAT* interface [26,11]. Using incremental SAT solvers avoids loading common subformulas over and over again. Moreover, it allows the solver to reuse

* This work was financially supported by the Academy of Finland, project 139402.

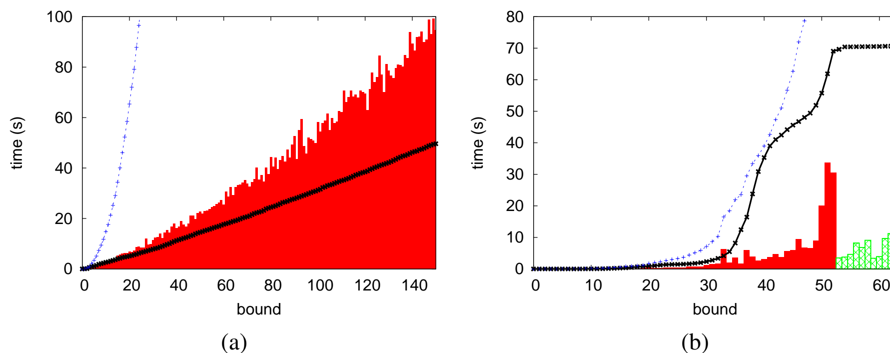


**Fig. 1.** Illustration of BMC run time behavior from [27]

the information it has gathered for consecutive problems. The resulting performance improvements make incremental SAT a crucial feature for modern SAT solvers.

One of the most common industrial uses of SAT solvers is in the area of formal verification. A particularly well established SAT based technique in this area is *Bounded Model Checking* (BMC) [4]. *Model checking* concerns proving temporal properties of systems, modelled e.g. as finite state machines. If a property does not hold for a system then this can be witnessed by a *counterexample*, which is a single valid execution of the system in which the property is falsified. Testing the existence of counterexamples of a bounded length can be easily done using SAT solvers. To achieve this, one defines an *unrolling function* which maps a formal system description, a temporal property, and an integer called the *bound* to a propositional logic formula. The unrolling function must encode the formula such that it is satisfiable iff a counterexample no longer than the given bound exists[1]. A typical BMC algorithm repeats this process starting from bound zero, and incrementing it by one as long as no counterexample is found.

Fig. 1 shows two illustrations of BMC run time behavior from [27], demonstrating the crucial impact of incremental SAT solving on BMC algorithm performance. The graphs illustrate solving time per bound for two different BMC benchmarks. The height of a bar in the graphs corresponds to the run time of a SAT solver on the formula for the corresponding bound without using incremental solving. The thick black curves illustrate the behavior of an incremental SAT solver that solved the formulas corresponding to all bounds sequentially, reporting its total run time each time it proceeded to the next formula in the sequence. The dotted blue curves are meant to further emphasize the poor performance of the non-incremental solver, by illustrating the cumulative run time of solving all formulas sequentially and independently.

Note that for the benchmark *eijk.S1238.S* illustrated in Fig. 1a the total run time for solving all bounds sequentially is only half that of solving the largest formula alone. Here, the gradual introduction of the problem to the solver has helped it to guide its search process, by "tuning" the solver on the smallest problems. Fig. 1b illustrates the behavior for benchmark *irst.dme6* for which the shortest counterexample is of length

[1] Another frequently used semantics is such that the formula is satisfiable iff the counterexample has a length *exactly* equal to the bound. This will be discussed in Sec. 3.4

53. The satisfiability of the formulas for bounds larger than or equal to 53 is emphasized by the hatched bars in the figure. Although solving only one of the satisfiable formulas using a non-incremental solver would be the fastest way of establishing the existence of a counterexample there is no way of telling in advance at what bound this "easy" problem resides. Advanced heuristics [23] for such predictions will be discussed in Sec. 3. For now, observe that the incremental solver provides a robust way of finding a counterexample without previous knowledge of its length.

Despite the importance of incremental solving for practical applications SAT solvers are typically benchmarked only on single formulas, both in research publications and during SAT solver competitions[2]. The community researching a different type of constraint solvers, called SMT solvers (*Satisfiability Modulo Theories*), has acknowledged the importance of incremental solving, by introducing the *application track* to their annual competition[3]. In that track solvers are tested on incremental problems [8].

Now that multi-core and multi-processor hardware has become widely available, parallelization of SAT solvers is actively researched [5,18,31,14,15,17]. Two major approaches can be distinguished. The first is the classic divide-and-conquer approach, which aims to partition the formula to divide the total workload evenly over multiple SAT solver instances [5,24,31]. The second approach is the so called *portfolio* approach [14]. Rather than partitioning the formula, portfolio systems run multiple solvers in parallel each of which attempt to solve the same formula. The system finishes whenever the fastest solver is done. Many such portfolios consist simply of multiple instances of the same CDCL solver, as such solvers can be made to all traverse the search space in different orders by as little as using different random seeds. Portfolio solvers thus mostly exploit the run time variance of different SAT solver runs on a single formula. This approach can be surprisingly effective. Parallel SAT solvers of both types can be extended with exchange of learnt clauses between SAT solver instances, which can greatly improve the efficiency, even enabling occasional super-linear speed-ups. Both techniques are evaluated in detail in [16] and elements from both techniques are used in a recently published new technique [17,18].

To the best of our knowledge, none of the work on parallelizing SAT solvers considered maintaining the incremental features, making these parallelizations hard to apply in many practical applications. In [29] we introduced Tarmo, which at the time was only envisioned to be a special purpose parallel solver for BMC. In 2011 Tarmo competed in the Hardware Model Checking Competition (HWMCC11), where it won the new experimental multi-property and satisfiable liveness property tracks. The competing version can be seen as a parallelization of the minimalistic BMC algorithm implementation *aigbmc*[4]. The latest Tarmo version, released in October 2012, is the first version that is easy to integrate into existing applications. It can provide such applications with substantial performance improvements, without requiring them to be modified.

This work makes explicit the notion of *asynchronous* incremental SAT, a simple but crucial concept for combining incremental SAT and parallelism. It allows more efficient parallelizations of the solving process, and simplifies the construction of *multi-engined*

[2] `http://www.satcompetition.org`
[3] `http://smtcomp.sourceforge.net`
[4] Part of the AIGER 1.9 toolset, `http://fmv.jku.at/aiger`

tools. Multi-engined designs are commonly found amongst applications of SAT solvers. For example, the majority of model checkers[5] that competed at HWMCC11 fall in this category [25]. Such tools include implementations of several different algorithms (engines) over which the available computation resources are divided. Although this division can be implemented using a sequential interleaving of execution steps of the different algorithms, nowadays such tools often employ so called *coarse grained parallelization*. This means that the tools perform largely independent tasks in parallel.

A related work is *Simultaneous SAT* [19]. The interface of a simultaneous SAT solver is different from a conventional solver as for each formula in the input sequence a set of *proof objectives* can be given. This type of solver aims to prove or disprove all of these proof objectives simultaneously, i.e. in a single backtracking search. The developers of simultaneous SAT intended it to be used for BMC algorithms that check multiple safety properties per bound. Unlike our approach simultaneous SAT requires modifying the search process of the solver. Using our asynchronous interface the behavior of a simultaneous solver can be simulated, and even parallelized. A simultaneous solver with an asynchronous interface can be envisioned, but has not been investigated.

## 2 Incremental SAT

In order to define and discuss incremental SAT in detail this section starts with some basic definitions. A *literal* $l$ is either a Boolean variable $x$ or its negation $\neg x$, and double negations cancel out, hence $\neg\neg l = l$. An *assignment* is a set of literals $A$ such that if $l \in A$ then $\neg l \notin A$. The assignment $A$ should be interpreted such that $l \in A$ means that $l$ is assigned the truth value **true**, and $\neg l \in A$ means that $l$ is assigned the truth value **false**. A *clause* $c$ is a set of literals $c = \{l_0, l_1, \cdots, l_n\}$ representing the disjunction $\bigvee c = l_0 \vee l_1 \cdots \vee l_n$. Hence, clause $c$ is *satisfied* by assignment $A$ iff $l \in A$ for some $l \in c$. Moreover, a clause consisting of exactly one literal is called a *unit clause*. A *cube* $d$ is a set of literals $d = \{l_0, l_1, \cdots, l_n\}$ representing the conjunction $\bigwedge d = l_0 \wedge l_1 \cdots \wedge l_n$. Hence, cube $d$ is *satisfied* by assignment $A$ iff $d \subseteq A$.

A formula is in *Conjunctive Normal Form (CNF)* if it is a conjunction of disjunctions, i.e. a set of clauses. A CNF formula is satisfied by an assignment that satisfies all of its clauses. A formula for which such a *satisfying assignment* exists is *satisfiable*, other formulas are *unsatisfiable*. Conventional SAT solvers handle only CNF formulas.

The most commonly used SAT solvers are of the *Conflict Driven Clause Learning* (CDCL) type [21]. Such solvers derive new clauses, called *learnt clauses*, during their solving process. These learnt clauses are logical consequences of the clauses in the input formula, and their derivation is intended to help the solver avoid parts of the search space that are without satisfying assignments. In this work the term solver always refers to a CDCL SAT solver for CNF formulas.

A general definition for the incremental satisfiability problem is given in [26], where it is defined as solving each formula in a finite sequence of formulas. The transformation from a formula to its successor in the sequence is defined by two sets, a set of clauses to be added and a set of clauses to be removed. Although it is possible to implement

[5] e.g. ABC [7] `http://www.eecs.berkeley.edu/~alanmi/abc`
and PdTRAV `http://fmgroup.polito.it/quer/research/tool/tool.htm`

a SAT solver that allows arbitrary removal of clauses between consecutive formulas, there is a complication in that when a clause is removed also all learnt clauses whose derivation depends on that clause must be removed. Maintaining sufficient information in the solver to achieve this has significant drawbacks on its performance and thus arbitrary clause removal is not implemented in any state-of-the-art solver.

Multiple solutions exists. For example, in the interface of the SAT solver *zChaff*[6], an implementation of the *chaff* algorithm [22], it is possible to assign clauses to groups, and those groups can be removed as a whole. The SMT-LIB standard [3] for SMT solver input defines the so called push- and pop-interface. In this approach the subproblems are maintained on a stack and the solver aims to solve the union of the problems on that stack. The simplest and most commonly used interface for incremental SAT solvers however is the one defined in [11] and first used in the solver *MiniSAT* [10]. This solver interface does not contain a function for removing clauses. Instead, a solver with this interface can determine the existence of satisfying assignments that include a specified set of *assumptions*. The interface is defined by two functions:

- `addClause(Clause clause)`
- `solve(Cube assumptions)`

Using this interface clause removal can be simulated as follows: Instead of adding clause $c$ to the solver the clause $c \cup \{x\}$ where $x$ is a free variable is added. As long as the solver is asked to perform its solving task under a set of assumptions that includes literal $\neg x$ it will only consider assignments $A$ such that $\neg x \in A$, hence it must satisfy $c$ in order to satisfy clause $c \cup \{x\}$. However, without the assumption $\neg x$ the solver can assign $x$ to **true** and ignore $c$.

Note that the `addClause` and `solve` function define part of the interface of a SAT solver, hence they control the execution of this particular computer program. The `solve` function is *blocking*, in the sense that the call to this function will not return to the calling application until the SAT solver determines the satisfiability of the loaded problem. In this work the input for an incremental SAT solver is defined separately from the execution of such a solver. Here, an instance of the incremental SAT problem is defined as a sequence of *jobs* $\langle \phi_0, \phi_1, \cdots \rangle$. A job $\phi_i$ is characterized by a set of clauses $\texttt{CLS}(\phi_i)$ and a single cube $\texttt{assumps}(\phi_i)$. Each job $\phi_i$ induces a CNF formula $\mathcal{F}(\phi_i)$ consisting of all its clauses and all clauses in previous jobs, and one unit clause for each literal in its cube of assumptions.

$$\mathcal{F}(\phi_i) = \underbrace{\left( \bigcup_{0 \le j \le i} \texttt{CLS}(\phi_j) \right)}_{\texttt{CLAUSES}(\phi_i)} \cup \left( \bigcup_{l \in \texttt{assumps}(\phi_i)} \{l\} \right)$$

In the rest of this work "solving a job" refers to the process of determining the satisfiability of the CNF formula induced by that job. Note that these definitions have been chosen to match solvers using the interface of [11]. Calling `addClause(c)` for

[6] `http://www.princeton.edu/~chaff/zchaff.html`

all $c \in \texttt{CLAUSES}(\phi_i)$ followed by a call to `solve`($\texttt{assumps}(\phi_i)$) will make such solver solve $\mathcal{F}(\phi_i)$ (assumptions are handled as truth assignments in the solver).

Without enforcing the blocking semantics of the `solve` function it is possible to think of the solver as a reactive system. The system is given jobs as input and as output it reports the result of solving those jobs. The communication between the application and the solver is *asynchronous*: The application may proceed to submit more jobs while the solver has not yet reported the result for a previously submitted job. Moreover, the results may be reported by the solver out-of-order with respect to the order of the jobs in the input sequence.

## 3 Employing asynchronicity and parallelism

To motivate the *asynchronous* communication between application and solver proposed in the previous section let us take another look at Fig. 1b. Note that the largest unsatisfiable formulas, those for bounds just below $53$, are much harder to solver than the smallest satisfiable ones. It was observed in [27] that this type of run time profile is typical for formula sequences from BMC that contain satisfiable formulas. This matched earlier observations [23] for a different application of SAT solvers called *automated planning*. In automated planning the satisfiability of a formula in the sequence corresponds to the existence of a *plan* of a certain length. The two applications are similar in nature: Either all formulas in the sequence are unsatisfiable, or the sequence has a finite prefix of formulas that are unsatisfiable, followed by only satisfiable formulas.

The authors of [23] did not consider incremental solving, but rather aimed to improve the speed at which the existence of a satisfiable formula in the sequence can be established using a non-incremental solver. They suggested that instead of always aiming to solve the first unsolved formula in the sequence, the total solving effort can be divided over a prefix of the unsolved formulas in the sequence. Under the observed typical run time profile this would then allow solving a satisfiable formula before the solving of the hardest unsatisfiable formulas has been completed. This is an interesting idea, but without the use of an incremental solver it is handicapped especially on long subsequences of unsatisfiable formulas. Although dividing the effort over multiple formulas can be beneficial, it is not useful if the extra performance provided by the incremental solver is lost. Asynchronicity provides a way to give an incremental solver any prefix of the formula sequence rather than just one formula at the time.

### 3.1 Parallelizing incremental SAT

The algorithms used in parallel SAT solvers for doing the actual solving are often identical to those used in sequential solvers. A typical parallel SAT solver's architecture uses multiple conventional sequential solvers in parallel. In portfolio solvers these parallel operating solvers are all given the same input, whereas in other approaches each solver instance is restricted to a portion of the search space. The basic building block in our parallel incremental SAT solver called Tarmo is a conventional incremental SAT solver using the assumptions interface, currently MiniSAT 2.2[7]. During its execution Tarmo

[7] `http://www.minisat.se`

spawns multiple *solver threads*, and each of these threads has access to its own instance of the conventional solver. Tarmo's interface is similar to that of any other SAT solver, except that it provides two extra non-blocking functions called `addCube` and `cancel`. The `addCube` function enters an assumptions cube, and thereby induces a new job in the sequence of jobs stored inside Tarmo. Each of its solver threads repeatedly reads a job from the sequence and solves it. The `cancel` function can be used to cancel the solving of a specific job.

If all of the solver threads always read the first unsolved job from the sequence then Tarmo becomes a portfolio of incremental solvers, e.g. each solver thread tries to solve all of the jobs in the input sequence. We refer to this strategy as *distribution mode* `multiconv` (multiple conventional). In a different distribution mode of Tarmo, called `multijob`, each of the solver threads always proceeds to solve the first unsolved job from the sequence that has not yet been assigned to another solver thread. This matches the natural idea that for an efficient parallelization the work performed by the separate threads should be different. This strategy was also used by a parallel solver specifically designed around one BMC unrolling function [1]. The `multijob` strategy does have a downside: Each solver thread individually no longer solves all of the jobs, hence the individual benefit of incremental solving is reduced.

As the solver threads use conventional incremental solvers no clauses can be removed by the solver threads. As a consequence, Tarmo can only use distribution modes which are defined such that a thread which just solved $\phi_i$ can only proceed to solve $\phi_j$ if $\texttt{CLAUSES}(\phi_i) \subseteq \texttt{CLAUSES}(\phi_j)$. Note that it is possible that $\texttt{CLAUSES}(\phi_i) = \texttt{CLAUSES}(\phi_j)$ for $i \neq j$ because applications may test the same set of clauses under different sets of assumptions. In such cases there are jobs $\phi_j$ such that $\texttt{CLS}(\phi_j) = \emptyset$. For example, in Cube-And-Conquer [15], one set of clauses is tested under many thousands of different sets of assumptions.

### 3.2 Clause sharing

Sharing of learnt clauses is an important building block in any parallel SAT solver. Although sharing learnt clauses between different solver threads can allow those threads to help each other, sharing too many clauses harms performance. Even conventional sequential solvers do not store all the learnt clauses they derive forever, but rather they clean up their learnt clause database regularly during the solving process. Restricting the number of learnt clauses shared between solving threads is therefore an important aspect of parallel SAT solving (see, e.g. [13]). It was stated in the introduction that incremental SAT solving "allows the solver to reuse the information it has gathered for consecutive problems". The learnt clauses are an important part of this information, although some heuristics measures kept in the solver are also important [27].

The asynchronous interface allows solving multiple jobs in any order. In particular, in Tarmo, multiple solver threads may not be solving the same job at the same time. Hence, care must be taken when employing sharing of learnt clauses between those solver threads. Note that in general a clause $c$ derived while solving a job $\phi_i$ can be used in the solving process of any job $\phi_j$ such that $\texttt{CLAUSES}(\phi_i) \subseteq \texttt{CLAUSES}(\phi_j)$.

To achieve correct clause sharing with low overhead the database in Tarmo is organized as a set of queues. There is one queue for each unique clause set, i.e. one queue

$q(\phi_i)$ for each job $\phi_i$ such that $\texttt{CLS}(\phi_i) \neq \emptyset$. For jobs $\phi_j$ such that $\texttt{CLS}(\phi_j) = \emptyset$ we have $q(\phi_j) = q(\phi_i)$ for the largest $i$ such that $i < j$ and $\texttt{CLS}(\phi_i) \neq \emptyset$. If a solver thread wants to share a learnt clause it derived while working at job $\phi_i$ it pushes it in the corresponding queue $q(\phi_i)$. A solver thread that is solving $\phi_j$ can now safely read and enter any foreign learnt clause that it can find in the queues $q(\phi_i)$ for all $i \leq j$.

The number of learnt clauses stored in each of the solver threads, and thus nominated for sharing with others, is not as massive in Tarmo as in conventional parallel SAT solvers for three different reasons. In Tarmo the solver threads only read and write to the queues in the shared clause database at the start and end of a job, and during *restarts* [12]. Some conventional solvers use a much more eager strategy. Sharing only at restarts however has the nice property that the introduction of new learnt clauses does not interfere with active search processes. The second reason is that the formulas used to test conventional parallelizations of SAT solvers are usually amongst the hardest its developer can find. Tarmo instead deals with sequences of problems for which the difficulty is typically more in the length of the sequence than in the hardness of individual formulas. The third reason is more implementation specific, but related to the second one. SAT solvers use a limit on the number of learnt clauses they store in their databases, and as the search continuous they increase this limit. A specific feature of MiniSAT, and thus also of the solving threads in Tarmo, is that when incremental solving is used this limit is reset for every consecutive call to `solve`. Hence, compared to solving a single hard instance for the same amount of time the clause database grows less large on an incremental problem sequence. During experiments for [15] this was found to be a crucial element in MiniSAT's incremental solving performance.

Unlike the common wisdom regarding conventional parallel SAT solvers, a version of Tarmo that shares *all* learnt clauses performs substantially better than the version that shares no clauses at all. Limiting the throughput of learnt clauses does improve its performance further, especially for harder problems. Tarmo limits the sharing of learnt clauses on the sending side only, i.e. clauses that are not considered of sufficient "quality" are not placed into the queues of the shared clause database. Two measures of clause quality that can be determined quickly are their length, and their Literals Blocks Distance (LBD) [2]. Because shorter clauses represent stronger constraints limiting the length of shared clauses by a constant (8 in [14]) would be a reasonable and very simple heuristic. The problem is that as the search continues the length of the clauses tends to increase, reducing the throughput of shared clauses [13]. Tarmo therefore by default shares all clauses whose length is below the running average, and this default is used in all results presented in this work. It is possible to configure Tarmo to share clauses below the average (or a constant) LBD, but this does not improve the average performance for the experiments presented here. The result of the experiments for different clause sharing heuristics can be found from the authors' webpage[8].

### 3.3 The synchronous interface: A drop-in replacement for MiniSAT

The aim of our work is to provide performance improvements for applications of incremental SAT solvers, without requiring extensive rewriting of those applications. To

[8] `http://users.ics.aalto.fi/swiering/tacas13`

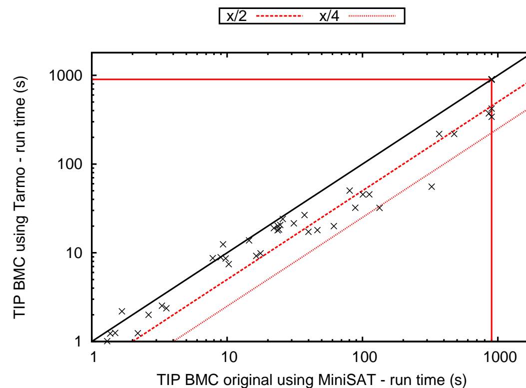


**Fig. 2.** Replacing MiniSAT by Tarmo without further modifications

illustrate that this can be achieved we took the latest version of the model checker TIP[9] and replaced the MiniSAT solver with Tarmo. Tarmo's interface provides a blocking `solve` call for full source-code compatibility with MiniSAT. Because of this compatible interface the modification of the source code of TIP was limited to just changing the name of the type of the solver. Although an application that uses Tarmo as a drop-in replacement for MiniSAT does not benefit from asynchronicity directly, it can still benefit from parallelism. Through Tarmo, and the `multiconv` distribution mode it provides, the application now has access to a portfolio of incremental solvers that are performing learnt clause sharing. Because most popular SAT solvers other than MiniSAT also use MiniSAT-like interfaces, replacing such solvers by Tarmo in existing applications should not be much harder.

All experiments in this work were performed in a computing cluster in which each node has two six core Intel Xeon X5650 processors. A memory limit of 3500MB per solver thread was employed. Fig. 2 is a logarithmic-scale scatterplot that shows the performance of the proposed straightforward use of Tarmo for the BMC algorithm inside TIP. This experiment was performed using the 95 benchmarks from the single safety property track of HWMCC11 for which during the competition at least one model checker found a counterexample. The version of TIP using the original MiniSAT solver solved 84 of those benchmarks within 900 seconds. By using Tarmo with 4 solver threads instead the performance of TIP is improved enough to make it solve 86 benchmarks. For the 24 benchmarks that were solved by the unmodified version of TIP in more than 10 seconds, an average speed-up of 2.1 is obtained by using Tarmo. A two time speed-up using four times the number of solver threads is not bad, considering that each of the solver threads are solving the exact same sequence of problems. During this experiment each of the solver threads used the exact same settings, except for the random seed. It should be possible to further increase the performance by using a variety of different settings for each solver thread, but this would require an extensive empirical evaluation that is outside the scope of this paper. The surprising strength of this approach matches observations for conventional parallel SAT solvers [14,18].

[9] `http://github.com/niklasso`

### 3.4 The asynchronous interface: Exploiting application specific knowledge

The asynchronous incremental solver interface is a natural extension to a basic incremental solver and can prove useful for many applications. Exploiting it effectively does however require some knowledge of the application.

The sequence of formulas generated from applications like BMC or automated planning can be generated up to any arbitrary length in advance. This does not hold for many other applications of incremental solvers in which the encoding of formulas depends on the results of solving previous formulas.

The main loop of a conventional BMC algorithm, as found in TIP, is given in Fig. 3a. The BMC unrolling function, providing the transition relation of a system for a bounded number of steps in propositional logic, is named `unroll` in the pseudocode. For this work it suffices to understand `unroll` as a function that makes repeated calls to the solver's `addClause` function and then returns a set of assumption literals. Once this has been done the `solve` function is called to establish the satisfiability of all clauses under the set of assumptions. If the solver finds this satisfiable then a counterexample of length $k$ has been found, otherwise the value of $k$ is incremented and the next iteration of the loop starts.

Fig. 3b illustrates a BMC loop exploiting the asynchronous solver interface. The non-blocking function `addCube` is called after `unroll`, inducing job $\phi_k$ for the solver. Note that $\mathcal{F}(\phi_k)$ is exactly the same formula that would have been solved in iteration $k$ of the conventional algorithm. On the Lines I-III the actions that must be executed when a result is received from the solver are stated. This result handling code can be executed in a thread concurrent to the thread executing the main loop, or alternatively it can be handled by the same thread if a poll to the solver for new results is included in the loop. In either case, Tarmo reports a result for each job $\phi_i$ at most once. For all but the most trivial benchmarks the encoding of a formula using the `unroll` function can be performed much faster than solving that formula. Hence, to avoid wasting large amounts of memory, in practice it is necessary to limit the number of unsolved jobs in the solver to a small constant. To illustrate this in Fig. 3b on Line 7 the job generation is paused until the value of shared variable $p$ falls below constant value $max_pending$. Alternatively, such limits can be implemented using functions provided by the interface of Tarmo, avoiding the need to handle potential concurrency issues in the application.

We modified TIP to use asynchronous BMC. TIP is a complex piece of software, which provides several different verification algorithms and performs non-trivial reductions on its input models. The modifications to the existing code of TIP made to introduce asynchronous BMC were, however, not more complicated than those given in Fig. 3. The performance is illustrated using a cactus plot in Fig. 4. The benchmarks used for the illustrated experiment are the same as discussed in Sec. 3.2. The two synchronous versions 'Sync. 1' and 'Sync. 4' correspond to the two algorithm versions compared in Fig. 2. Observe that using $4$ solver threads and Tarmo's `multijob` distribution mode, asynchronous BMC is able to solve $88$ of the benchmarks. Using $6$ threads this further increases to $89$, but it then goes back to $88$ for the version that uses $8$ threads.

Earlier in this work, and in the related work on automated planning [23], only sequences were considered that either consist only of unsatisfiable formulas, or of a finite

| Conventional BMC | Asynchronous BMC |
|---|---|
| 1. $k = 0$<br>2. `forever do`<br>3. $A =$ `unroll(k)`<br>4. $r =$ `solve(A)`<br>5. `if` $r =$ **unsatisfiable** `then`<br>6. $k++$<br>7. `else`<br>8. `return` cex of length $k$ | 1. $k = 0$; $p = 0$<br>2. `while` cex not found<br>3. $p++$<br>4. $A =$ `unroll(k)`<br>5. `addCube(A)`<br>6. $k++$<br>7. `wait until` $p < max_pending$<br>On result for job $\phi_i$:<br>I. $p--$<br>II. `if` result for $\phi_i$ is **satisfiable** `then`<br>III. `return` cex of length $i$ |
| (a) | (b) |

**Fig. 3.** Pseudocode for usage of incremental SAT in BMC

prefix of unsatisfiable formulas followed by only satisfiable formulas. This means that the result handling functions of Fig. 3b can be extended with an extra application specific improvement: If the result unsatisfiable is reported then the solver may be asked to abort solving all unsolved older jobs, as these are now known to be unsatisfiable. The `cancel` function in Tarmo's interface is provided for this purpose. Unfortunately there is a problem when applying this idea in TIP, which is that for safety properties it encodes the $k$-th formula with the semantics that it is satisfiable iff a counterexample of *exactly* length $k$ exists. Hence, in TIP, the unsatisfiability of a job does not necessarily imply that all older jobs are also unsatisfiable.

This problem was resolved by making a small modification to each of the benchmarks before giving them as input to our asynchronous BMC version of TIP. The benchmarks are encoded in the AIGER-format[10], a representation of Boolean circuits using and-gates, inverters and latches. Here, a counterexample is a sequence of truth assignments to the inputs of the circuit that makes the output attain the value **true**. For each benchmark a new circuit was created by extending the original circuit with a small amount of extra logic, including one latch. The added logic makes sure that, iff the output of the original circuit attains the value **true**, then the output of the new circuit attains the value **true** and remains in this state regardless of changes to the input signals. By using these modified circuits, instead of the original models, older jobs can now be safely cancelled by the asynchronous BMC result handling function. The resulting performance is shown in Fig. 5. Clearly, cancelling of older unsatisfiable jobs improves the performance and especially the scaling of the parallelization.

### 3.5 Coarse grained parallelization

For computationally hard problems, such as SAT solving or model checking, there are no "one size fits all" solutions. Because different algorithms work well for different

[10] `http://fmv.jku.at/aiger`

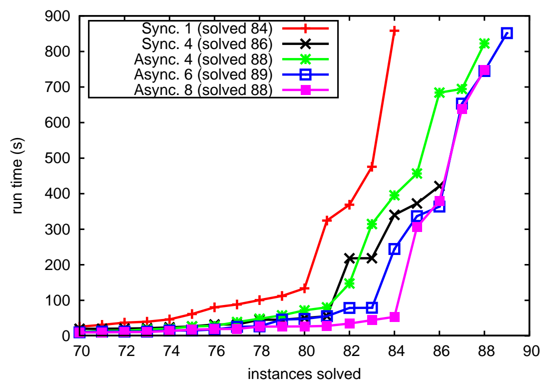


**Fig. 4.** TIP BMC using asynchronous solving

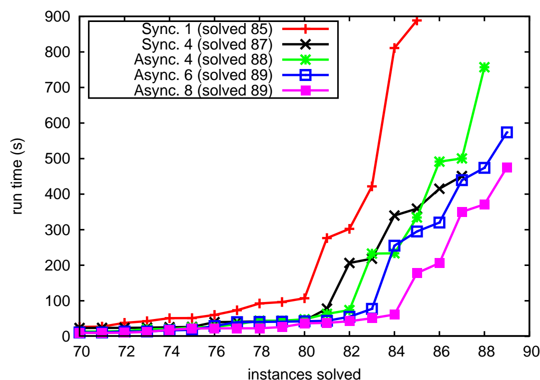


**Fig. 5.** TIP BMC using asynchronous solving on modified circuits

problems, tools implementing more than one algorithm, so called *algorithm portfolios*, or *multi-engined tools*, are common practice (e.g. [30,25]). Although the asynchronous interface was developed to allow parallelization of incremental SAT solving, it can also aid the development of multi-engine tools. Once again, we use TIP to illustrate our point. TIP includes an implementation of the IC3 algorithm [6] which is called the *Recursive Induction Prover (RIP)*. In contrary to the basic BMC implementation this algorithm can prove that a property holds. Although IC3/RIP can also find counterexamples it can typically not match the performance of BMC at this task, thus executing both algorithms in a portfolio should provide better average performance.

Creating such a portfolio inside TIP was easy, as we had asynchronous BMC already in place. We simply added calls to the BMC algorithm functions `unroll` and `addCube` (recall Fig. 3b) inside the main loop of the RIP algorithm. As a result, the RIP algorithm ensures the concurrent execution of the completely independent asynchronous BMC algorithm. In this set-up Tarmo is only used for BMC. Using the RIP algorithm $346$ out of the $465$ single safety property benchmarks from HWMCC11 can be solved within $900$ seconds. By executing BMC concurrently with RIP this increases

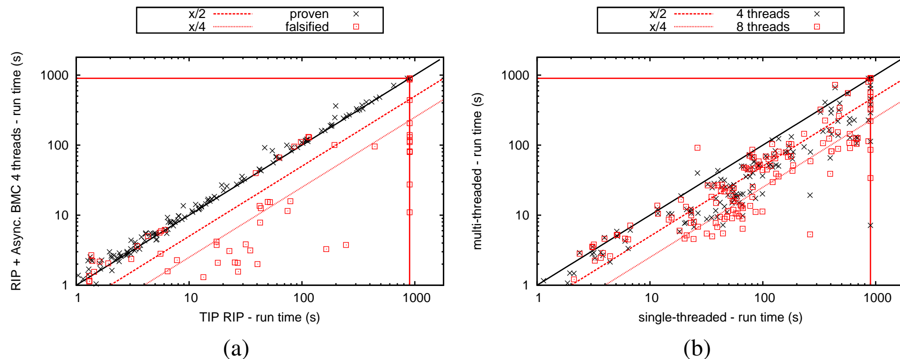


**Fig. 6.** Tarmo for concurrent BMC and RIP, and Tarmo for MUS Finding

to 352, with Tarmo configured to use one solving thread for BMC. Using 4 solving threads 357 benchmarks are solved. The impressive and consistent speed-up for counterexample finding is illustrated in Fig. 6a for the version using 4 solving threads.

It must be noted that simultaneous execution of two completely separated implementations of BMC and RIP as two different processes will give roughly the same performance. This experiment is only meant to illustrate that an asynchronous solver is easy to run "on the side". This clearly can have advantages over execution in separate processes. For example, one could implement a tool in which the BMC and RIP algorithms share derived system invariants, or lower-bounds on counterexample length.

### 3.6 Asynchronous solving outside BMC

Some applications of incremental solvers, such as Cube-And-Conquer [15] parallelize naturally, whereas others are very challenging. Dependencies between the generation of jobs and the result of previous jobs can make running multiple jobs concurrently harder. In this section we discuss a particularly challenging application.

An unsatisfiable CNF formula is *minimal unsatisfiable* if removing any of its clauses makes it satisfiable. Algorithms that find Minimal Unsatisfiable Subsets (MUSes) of unsatisfiable formulas have received a lot of research interest in recent years. An important recent contribution is *model rotation* [20]. The performance of that algorithm was studied in [28], which also proposed parallelization using Tarmo. This a challenging application because the concurrently executed jobs are not independent. In this parallelization the result of a job can imply that the result of concurrently solved jobs is no longer interesting. Fig. 6b shows results for a new implementation of the existing parallelization from [28]. The new implementation is based on the same ideas but benefits from Tarmo's recent interface improvements, as well as from better MUS finding heuristics. The set of benchmarks used were the 178 benchmarks also used in [28] and 34 from [4]. The single threaded version solved in total 168 benchmarks, requiring on average 2468 jobs per benchmark. The versions using 4 and 8 threads both solve 174 benchmarks. However, the 4 threaded version opportunistically generates an average of 3610 jobs per benchmark out of which only 2499 (69%) have a result that progresses the MUS finding. For the 8 threaded version only 2535 (52%) out of 4842 jobs per

benchmark are effective. Despite the large amount of unnecessary work performed, this parallelization improves the performance of a state-of-the-art MUS finding algorithm.

## 4 Conclusions

In this paper we discussed the *asynchronous* interface for incremental SAT solvers. The incremental feature of modern SAT solvers is crucial for their performance in practical applications. Nevertheless, it is often overlooked in research aiming at improving or parallelizing such solvers. By extending the most commonly used incremental solver interface our parallelizations are directly applicable in many different contexts. As a result, substantial performance gains can be obtained by simply replacing a sequential incremental solver by our source-code compatible multi-core solver. In many cases further improvements are possible by using the asynchronous interface to create an application specific parallelization. The minimal nature of the proposed extension to the standard interface means that asynchronicity does not have to be limited to our Tarmo solver. Instead, it can prove useful to any solver developer interested in combining incremental SAT solving and parallelism.

**Acknowledgements** The authors would like to thank Niklas Sörensson for providing TIP, Niklas Eén and Armin Biere for being sources of motivation and inspiration, and Matti Niemenmaa for his contributions to early versions of Tarmo.

The Tarmo solver, the modified versions of TIP, and more experimental data can be found from `http://users.ics.aalto.fi/swiering/tacas13`.

## References

1. Ábrahám, E., Schubert, T., Becker, B., Fränzle, M., Herde, C.: Parallel SAT solving in bounded model checking. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS/PDMC. Lecture Notes in Computer Science, vol. 4346, pp. 301–315. Springer (2006)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI. pp. 399–404 (2009)
3. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB standard version 2.0 (2010), `http://www.smtlib.org`
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, R. (ed.) TACAS. Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer (1999)
5. Böhm, M., Speckenmeyer, E.: A fast parallel SAT-solver - efficient workload balancing. Ann. Math. Artif. Intell. 17(3-4), 381–400 (1996)
6. Bradley, A.R.: k-step relative inductive generalization. CoRR abs/1003.3649 (2010)
7. Brayton, R.K., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV. Lecture Notes in Computer Science, vol. 6174, pp. 24–40. Springer (2010)
8. Bruttomesso, R., Griggio, A.: Broadening the scope of SMT-COMP: the application track. In: COMPARE. pp. 18–27 (2012)

9. Cook, S.A.: The complexity of theorem-proving procedures. In: STOC. pp. 151–158. ACM (1971)
10. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT. Lecture Notes in Computer Science, vol. 2919, pp. 502–518. Springer (2003)
11. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
12. Gomes, C.P., Selman, B., Crato, N., Kautz, H.A.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J. Autom. Reasoning 24(1/2), 67–100 (2000)
13. Hamadi, Y., Jabbour, S., Sais, L.: Control-based clause sharing in parallel SAT solving. In: Boutilier, C. (ed.) IJCAI. pp. 499–504 (2009)
14. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: A parallel SAT solver. JSAT 6(4), 245–262 (2009)
15. Heule, M.J., Kullmann, O., Wieringa, S., Biere, A.: Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In: Haifa Verification Conference (HVC) (2011)
16. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. In: Serra, R., Cucchiara, R. (eds.) AI*IA. Lecture Notes in Computer Science, vol. 5883, pp. 243–252. Springer (2009)
17. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Grid-based SAT solving with iterative partitioning and clause learning. In: Lee, J.H.M. (ed.) CP. Lecture Notes in Computer Science, vol. 6876, pp. 385–399. Springer (2011)
18. Hyvärinen, A.E.J., Junttila, T.A., Niemelä, I.: Partitioning search spaces of a randomized search. Fundam. Inform. 107(2-3), 289–311 (2011)
19. Khasidashvili, Z., Nadel, A., Palti, A., Hanna, Z.: Simultaneous SAT-based model checking of safety properties. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) Haifa Verification Conference. Lecture Notes in Computer Science, vol. 3875, pp. 56–75. Springer (2005)
20. Marques-Silva, J.P., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT. Lecture Notes in Computer Science, vol. 6695, pp. 159–173. Springer (2011)
21. Marques-Silva, J.P., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD. pp. 220–227 (1996)
22. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: DAC. pp. 530–535 (2001)
23. Rintanen, J., Heljanko, K., Niemelä, I.: Planning as satisfiability: parallel plans and algorithms for plan search. Artif. Intell. 170(12-13), 1031–1080 (2006)
24. Schubert, T., Lewis, M.D.T., Becker, B.: PaMiraXT: Parallel SAT solving with threads and message passing. JSAT 6(4), 203–222 (2009)
25. Sterin, B., Een, N., Mishchenko, A., Brayton, R.: The benefit of concurrency in model checking. In: IWLS. pp. 176–182 (2011)
26. Whittemore, J., Kim, J., Sakallah, K.A.: SATIRE: A new incremental satisfiability engine. In: DAC. pp. 542–545 (2001)
27. Wieringa, S.: On incremental satisfiability and bounded model checking. In: Ganai, M.K., Biere, A. (eds.) DIFTS. pp. 46–54 (2011)
28. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) CP. Lecture Notes in Computer Science, vol. 7514, pp. 672–687. Springer (2012)
29. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: Brim, L., van de Pol, J. (eds.) PDMC. EPTCS, vol. 14, pp. 62–76 (2009)
30. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. CoRR abs/1111.2249 (2011)
31. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symb. Comput. 21(4), 543–560 (1996)