QuAPI: Adding Assumptions to Non-Assuming SAT & QBF Solvers

Maximilian Levi Heisinger[1], Martina Seidl[1] and Armin Biere[2]

[1] *Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria*
[2] *University of Freiburg, Friedrichstr. 39, 79098 Freiburg, Germany*

Abstract
We present QuAPI, an easy to use library for enabling efficient assumption-based reasoning for arbitrary SAT or QBF solvers. Our library allows applications of SAT or QBF to benefit from assumptions and thus incremental solving matching the native support of a standard incremental interface such as IPASIR, but without the need to implement it. This feature is particularly valuable for applications of QBF solving as most state-of-the-art solvers do not support assumption-based reasoning, with some not providing any API at all. The users of QuAPI only need access to an executable of these solvers. The only requirement on the solvers is to use standard C library functions, either wrapped or directly, for reading input files in the standard DIMACS or QDIMACS format. No source code or library access is necessary, also not for API-less solvers. In this paper, we introduce the architecture of QuAPI and discuss crucial implementation details in depth. Our extensive performance evaluation shows substantial speedup compared to using unmodified solver binaries on similar incremental reasoning tasks.

Keywords
SAT, QBF, Assumption-Based Reasoning

1. Introduction

In SAT solving, assumption-based reasoning is essential for the extraction of minimum unsatisfiable sets (MUS) [1], the computation of minimal correction sets (MCS) [2], the realization of cube-and-conquer (CnC) solving [3] or for applications like bounded model checking (BMC) [4] to name only a few examples. In all these tasks, it is necessary to successively solve many similar SAT problems which are distinguishable only by assumptions as introduced by Eén and Sörensson [4]. Concrete use cases of assumptions are the enabling/disabling of certain clauses in different solver runs or the generation of subproblems in which some variables are already assigned a value.

Assumptions are literals that, in one particular solver run, enforce the assignment of the respective variables according to their polarity. To this end, the assumption literals either impose the first decisions of the solver or they are temporarily added as unit clauses

PAAR'22: 8th Workshop on Practical Aspects of Automated Reasoning, August 11–12, 2022, Haifa, Israel
EMAIL: maximilian.heisinger@jku.at (M. L. Heisinger); martina.seidl@jku.at (M. Seidl); biere@informatik.uni-freiburg.de (A. Biere)
URL: https://maxheisinger.at (M. L. Heisinger)
ORCID: 0000-0001-7297-6000 (M. L. Heisinger); 0000-0002-3267-4494 (M. Seidl); 0000-0001-7170-9242 (A. Biere)

and are dismissed after the formula has been decided. Therefore, they are not available in the next solver run and new assumptions can be provided to the solver. Employing a solver that supports assumption-based reasoning is not only very programmer-friendly and convenient from a usability perspective. It also makes the repeated re-parsing of the same clause set obsolete, thus significantly reducing this unnecessary overhead. Furthermore, certain repetitive inference steps can be saved. Already introduced by MiniSat [5], nowadays many SAT solvers provide a programmatic interface to support SAT solving under assumptions and several approaches have been suggested for their efficient implementation [6, 7, 8, 9].

Also for applications of *quantified Boolean formulas* (QBFs), the extension of SAT with quantifiers over the propositional variables, the value of assumption-based reasoning has been early recognized [10]. While there has been enormous progress in QBF solving technology [11] witnessed by a rich solver landscape [12], to the best of our knowledge, only the search-based QBF solver DepQBF [13] supports assumptions. During the development of QBF applications, we found this a severe restriction considerably reducing our user experience when applying modern QBF solvers. This turned out to be especially problematic in projects on incremental model counting, the extraction of minimal unsatisfiable cores, and our extension of the CnC-solver Paracooba [14] to reasoning on QBF.

In order to overcome this restriction, we developed QuAPI which provides an easy-to-use interface for assumption-based reasoning even for solvers that do not support any kind of incremental solving. The requirements on a solver to be used within our QuAPI framework are very small: its binary needs to be available and it has to use standard C library functions (or wrap them, e.g. with C++ iostreams) for reading input files in the DIMACS or QDIMACS format. Other library functions may also be overriden, with e.g. zlib's `gzread` (used by MiniSat) being supported as-well. Neither source code nor any library access to the solver are necessary. Not requiring API access to solvers is essential, as some state-of-the-art solvers (e.g. Caqe [15] and RAReQS [16]) do not offer such interfaces. By instrumenting arbitrary binaries of SAT and QBF solvers, we provide a user-friendly and efficient solution. This enables the direct integration of broader solver portfolios by implementing against an established style of C-API, without requiring specific source-code (or other) extensions.

In this paper, we first introduce QuAPI by an example demonstrating its simple C-API. We then describe its architecture and implementation in detail. Finally, extensive experiments show QuAPI's performance compared to alternative means of interfacing with solvers. Our library and the Quapify tool which enables giving assumptions as command-line arguments to generic solvers, are publicly available at

https://github.com/maximaximal/quapi

2. Related Work

Originally introduced by the SAT solver MiniSat [5], incremental SAT solving with assumption-based reasoning is nowadays implemented in many SAT solvers. To establish a standard, a simple C interface for incremental SAT solvers is specified by the IPASIR

```
quapi_solver *s = quapi_init("/usr/local/bin/caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumption vars n */
                             NULL, /* SAT regex */
                             NULL  /* UNSAT regex */);
quapi_quantify(s, -1); /* ∀x1 */
quapi_quantify(s, 2); /* ∃x2 */
quapi_add(s, 1), quapi_add(s, -2), quapi_add(s, 0); /* x1 ∨ ¬x2 */
quapi_add(s, -1), quapi_add(s, 2), quapi_add(s, 0); /* ¬x1 ∨ x2 */
quapi_assume(s, 1); /* assume x1 to be true */
int status = quapi_solve(s); /* wait for results */
assert(status == 10); /* solved! */
quapi_assume(s, -1); /* assume x1 to be false */
status = quapi_solve(s); /* wait for results */
assert(status == 10); /* solved again! */
quapi_release(s); /* release resources at the end */
```

Listing 1: Usage example.

(Reentrant Incremental Sat solver API, in reverse) standard [17] first used in the SAT competition's incremental track of 2015. The main design goals behind IPASIR are that it is both easy to use and not too complex to implement efficiently, while also at the same time covering most usage scenarios of incremental SAT solving. The implementation of the interface remains in the responsibility of the solver developers. A different approach is followed by the PySAT [18] framework. This framework integrates several recent SAT solvers into one library such that they can be used uniformly via a Python interface. If a new solver is added, then the framework has to be updated respectively.

We are not aware of any unifying standard or framework for incremental QBF solving. An incremental extension was suggested almost a decade ago in the QBF solver QuBE [10], but this solver is not available any more. The only recent solver that provides an interface for incremental solving is the search-based QBF solver DepQBF [13]. Assumption-based reasoning is deeply integrated into the solving process and required substantial changes to the solver's internal cube and clause learning. Also the solver state has to be reset in each incremental run.

3. The QuAPI Library

Before we discuss the architecture and implementation of QuAPI in detail, we first illustrate its usage by a simple example.

3.1. QuAPI By Example

Given the QBF $\forall x_1 \exists x_2((x_1 \lor \neg x_2) \land (\neg x_1 \lor x_2))$, we want to solve this formula first under the assumption that x_1 is true and afterwards under the assumption that x_1 is false. The solving is done with the recent QBF solver Caqe [15] which is implemented in the programming language Rust and does not provide an incremental interface. Listing 1 shows the code of our example. First, the solver is registered and initialized by calling the function `quapi_init`. Besides the path to the solver binary and optional command-line and environment parameters, the variable and clause number of the formula to be solved are passed to the solver (in our case we have two variables and two clauses). Further, the maximum number n of assumption variables is provided. For QBF, the first n variables in the prefix are then considered for the assumptions. For SAT, which does not impose any ordering on the variables, there is no restriction on the variables which are used for assumptions. Next, the prefix of the QBF is added followed by the clauses. By the call of `quapi_assume (s, 1)` variable x_1 is assumed to be true (note that the QDIMACS format uses integers for variables names). The call to the solver tells us that the QBF is true under the respective assumption. Next we assume x_1 to be false. Also here the solving result is true as expected. To release resources we call `quapi_release`. This example illustrates how easy assumption-based solving becomes when using QuAPI even if the solver does not support assumptions. For the user, it is just like calling a solver library that supports IPASIR-style assumptions. In the background, however, a lot of things are happening. We discuss the implementation details of QuAPI in the following.

3.2. Architecture and Implementation of QuAPI

Figure 1 shows the general architecture of QuAPI as well as the information flow. In the following we reference the blue numbers of Figure 1 in order to make the data and control flow easier to follow. When the user registers a solver with `quapi_init`, the application process starts a new factory process via `fork` *(1)*. Then the local environment variables (of the host system) are modified such that they contain `LD_PRELOAD=<path to quapi_preload.so>`. If the library was built with enabled address sanitizer (CMake Option `DEBUG_ENABLE_ADDRESS_SANITIZER`) for debugging, the build system is automatically queried for the correct path to *libasan* to also be added to `LD_PRELOAD`, such that the whole system may be run using the specified sanitizer. In this way, the factory and solver processes have all required symbols for providing debugging information. A call of `execvpe` starts the solver. Next, the shared object is loaded and *ld* overwrites other symbols otherwise loaded from the system's *libc*. Most importantly, `read` and alternatives (`getc`, `_unlocked` variants, etc.) are overwritten in order to gain control over the solver's input. Once the solver binary starts up, it tries to read from its standard input, calling QuAPI's `quapi_read` instead of the standard implementation provided by the operating system. The original symbols are resolved during the preloaded QuAPI runtime's initialization using `dlsym` and are executed if the solver reads from other files than standard input.

After QuAPI's runtime has been initialized, it reads control messages in its proprietary binary format from standard-in (*STDIN*). Commands are sent by function calls of the

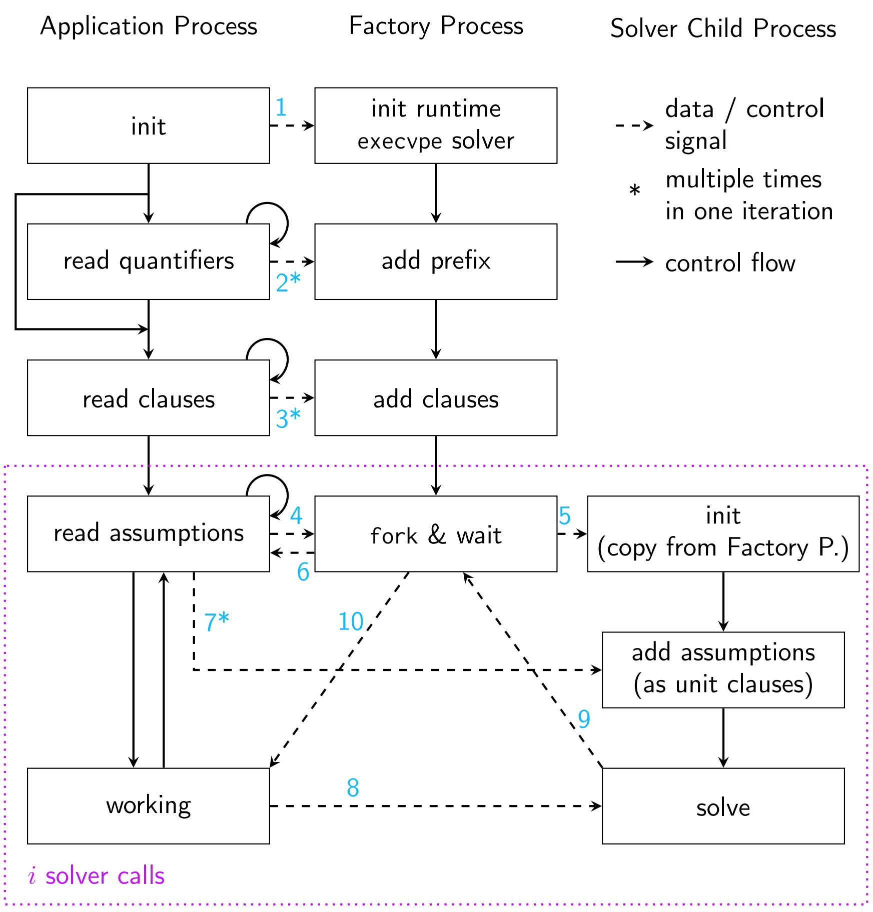

Figure 1: The QuAPI processes and their message exchange (numbers indicate messages ordering).

user-facing library in the application process. The problem header of the specific formula instance is sent first in form of a `PROBLEM_HEADER` message containing variable and clause counts and pipe file-descriptors (*fds*) for communication with the parent process. It also contains an API version identifier, which is matched against the one in the dynamically loaded runtime in order to keep QuAPI expandable. Also the maximum number of assumptions is provided. The runtime in the factory process reads the header message and generates a valid (Q)DIMACS header string `p cnf <vars> <clauses>`, which it submits into the `read` buffer of the already running solver. In this problem header the number of clauses is increased by the maximum number of assumptions. At this point, the control is handed over to the solver, which continues with the provided data as if it was read from some file or pipe.

Next, the application process sends the formula to the solver. In case of QBF, it begins with the quantifier prefix (multiple calls of `quapi_quantify`) *(2)*. If n is the maximum number of assumptions, then the first n variables of the prefix are considered to be assumptions and are therefore existentially quantified. This is necessary, because fixing a universally quantified variable by a unit clause would change the semantics of the formula and render it trivially false. As consequence, universally quantified assumption variables may not be left unassigned. After the optional quantifier prefix, the formula matrix is sent *(3)*. Literals are stringified efficiently using the `format_int` algorithm adapted from the {fmt} library [19]. When the processing of the formula is finished, assumptions may be added. To this end a `FORK` message is issued *(4)*. Control in the factory process returns to the QuAPI runtime again. The factory process is now forked another time *(5)*, forming a solver child process. The user-facing library in the application process waits to receive a `FORK_REPORT` from the factory process *(6)*, containing the file descriptor to write the assumptions. They are then transformed into unit clauses, which are sent from the application process to the solver child process *(7)*, completing the formula with the assumption unit clauses. The original version of the solver in the factory process is not changed during this, saving a copy of the solver initialized with the original formula.

Once a `SOLVE` message is sent *(8)* by calling `quapi_solve`, the receiving solver child process checks if enough clauses have been written to match the problem header given at the beginning. If there are still clauses missing, the solver child automatically generates dummy clauses in form of $(x \lor \neg x)$, with x being 1 for propositional formulas or the first variable from the quantifier prefix in PCNF formulas and feeds them to the solver. When the missing clauses have been provided, `EOF` signals the solver to begin solving. The factory process now waits for a return code *(9)* using `waitpid`. Once the solver child has finished, the factory process records its return code and sends it as `EXIT_CODE` back to the application process *(10)*, which waits for this result (or an abort) using `poll` on the respective file descriptors. It receives the result and returns it normally from `quapi_solve`. As indicated in purple, the `quapi_assume` and `quapi_solve` flow can be repeated, each time solving the original formula under a different assumption.

In addition to using `waitpid` and extracting a solver's result from its return code, one may provide SAT and UNSAT detection regular expressions. If these are specified, QuAPI also waits for data from standard output of the process' grand-child. If no SAT or UNSAT regular expressions are specified, standard output is not processed further. Lines are read into a buffer and processed by the PCRE regular expression library, if the library was available at build time. Using this output parsing mechanism, also solvers that do not follow the established SAT-solver return code convention (10 meaning SAT and 20 meaning UNSAT) may be used, without additionally writing wrapper scripts around solver binaries. This parsing mechanism is disabled if PCRE is not found by the build system.

The user-facing QuAPI library behaves like a regular solver interface during the whole interaction, creating processes and waiting for results without further management required.

4. Performance Evaluation

Our QuAPI library heavily interferes with the solver's I/O. To assess the impact of our modifications we performed extensive experiments. With the aim to benchmark QuAPI's

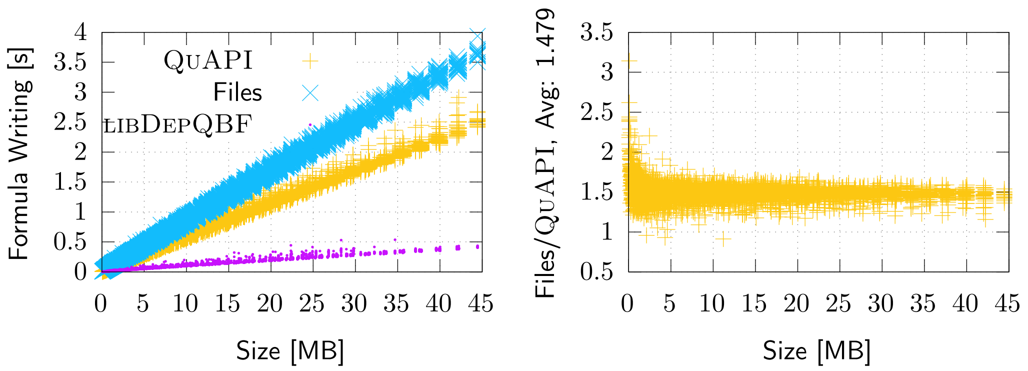

Figure 2: Solver initialization times for formulas of different sizes (left) and overhead of file-based approach compared to QuAPI (right).

performance while minimizing the solver-specific impact, we generated 8400 random 3-SAT instances parameterized over their literal-to-clause ratio (LCR), literal count l, and total target clause count c. We then prefixed these 3-SAT formulas with randomly ordered, existentially quantified variables. In order to generate n different problems with assumptions from one formula, in each such problem we assume the first n variables to be false, except for one (varying with the index of the assumption), which is true. These sub-formulas are (purposefully) meaningless, in order for the solve-times of solver calls to be as equally distributed as possible, thus enabling a fair comparison between the employed communication methods. We decided to use 3-SAT and not 3-QSAT for our experiments, to get a uniform setup for the performance evaluation of QuAPI.

For LCR we considered the range 3.27 to 3.47 with 0.01 increments, for l the range 100 to 2000 with 100 increments, and for c the range 20 to 400 with 20 increments. These ranges showed a good balance between time to completion and observable behavior. We obtained formulas of up to 45 MB. We performed our tests with 1, 2, 4, 8, and 16 solver calls under assumptions. In the following, we focus on the latter case. As DepQBF is the only QBF solver with an incremental interface, we used this solver for our experiments. We compare three variants: (1) libDepQBF in which we programmatically use assumptions via the incremental interface of DepQBF, (2) DepQBF with files such that for every solver call a file with the formula and the assumptions is written and (3) QuAPI as a wrapper of the DepQBF binary. All benchmarks were run on Ubuntu 20.04 with dual-socketed Intel(R) Xeon(R) E5645 CPUs @ 2.40 GHz (six cores) and 94 GB RAM, with six benchmark instances in parallel.

4.1. Solver Initialization

In our first experiment, we are interested in comparing the costs of providing the input formula to the solver. Therefore, we consider the time it takes to provide an already parsed formula either via API calls to libDepQBF or QuAPI or to write it to a file in `/dev/shm`. So far, no assumptions are involved in this setup. The outcome of our

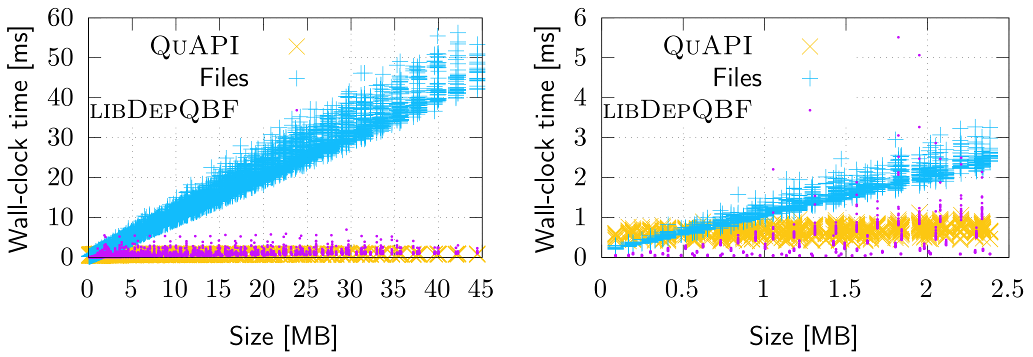

Figure 3: Runtimes to apply all assumptions. Graph on the right is zoomed in to the very beginning of the full dataset displayed on the left.

experiments is summarized in Figure 2. We see that QuAPI is faster than the file-based approach, but about five times slower than using libDepQBF directly. This overhead is not surprising, because the solver called by QuAPI is only given a textual representation of the formula, has to receive it from a different process and also has to parse it.

Performance parity between the file-based approach and QuAPI, i.e. both being equally fast at solving just one assumption, was reached by using buffered reading and writing internally. This was implemented using `fread` and `fwrite`, which decreased the formula transfer time by a factor of three compared to directly relying on the (unbuffered, direct IO) system calls `read` and `write`. The only downside of buffered IO is the requirement to flush the write-end of the pipe depending on messages sent, as `FORK` and `SOLVE` messages must be sent immediately in order to not break the underlying state machine. If formulas contain more than 1000 variables, QuAPI's inter-process communication uses less data than direct file-writing, as one literal in the internal message format takes 5 bytes to transfer, while the overhead increases further in text-based (Q)DIMACS.

Additionally to buffered IO, we tried using `vmsplice` on the write end in conjunction with `splice`, `memfd`, and `mmap` on the read end to reduce the number of required copies between user-space and kernel-space. This improved the throughput when writing formulas and improved the overall speedups by a bit (around 2 % over the whole benchmark set), but also increased the time it takes to apply assumptions. We therefore do not enable this (almost) zero-copy implementation by default, but instead offer it as the compile-time CMake option `ENABLE_ZEROCOPY`. This is useful when solving very big formulas with few assumptions.

4.2. Addition of Assumptions

For evaluating the performance of applying assumptions to a formula and solving it using a regular (unmodified) solver binary, every set of assumption literals is added as unit clauses to a copy of the original formula. We consider the addition of 16 sets of assumption literals, i.e. we call the solver 16 times on each benchmark under different assumptions. The newly generated files contain updated problem headers, the whole original formula, and appended clauses, one unit clause for each added assumption. Assumption files also are written to `/dev/shm` in order to efficiently communicate with the solver binary as if it would use shared memory. We compare this file-writing to using QuAPI for applying assumptions to the same solver binary, and to using the libDepQBF API. We consider the time it takes to add one assumption. Solving time is not included.

The results of this experiment are shown in Figure 3. File writing leads to a linear run-time overhead, while both QuAPI and the DepQBF library stay about constant. Here QuAPI clearly benefits from its `fork` approach which omits writing a new file per assumption. Small files less than 0.5 MB tend to be written into shared memory faster than observed with the `fork` mechanism as implemented in QuAPI, but with increasing size, file writing does not scale as well while the `fork` overhead stays constant. This timing is not perfect though, as the QuAPI-internal buffered write flushes with a `FORK` message, which is sent when applying the first assumption. The real time it takes for assumptions to be applied therefore decreases a bit further. This makes QuAPI comparable to using a solver library directly, while still being generically applicable to any solver binary.

4.3. Overall Runtime

In our final experiment, we evaluate the impact on the overall wall-clock time. The time is calculated by summing up initialization, initial formula writing, summed assumptions, and summed solve times for all applied assumptions combined. Speedup from t_1 to t_2 are calculated using $\frac{t_1}{t_2}$, always comparing file-based execution of a solver binary (Exec) with QuAPI. We observe smaller speedups for harder to solve formulas, which we show by scaling formula hardness using multiple LCR values. Randomly generated assumptions on random 3-SAT formulas have similar solving times between each different set of assumption literals on the same formula, removing potential influences of changing difficulty between different assumption sets in our benchmarks. The speedups get larger when using QuAPI with assumptions that measurably simplify a problem, as less time is spent in each full solver run.

Figure 4 summarizes the connection between the time it takes to solve a formula and its size. Increasing formula size leads to larger speedups, as assumption application and formula parsing times stay constant. Increasing formula hardness decreases effective speedup, as more time is spent solving the formula compared to the time taken to continuously re-process the same formula for every assumption. Changing the literal count does not influence this relation, but higher literal counts make it more pronounced.

5. Conclusion and Future Work

We developed the library QuAPI which provides a user-friendly API to equip arbitrary SAT and QBF solver binaries with assumption-based reasoning. In our extensive evaluation we could show that QuAPI not only improves usability compared to a file-based solution, but also shows substantial performance improvements.

With QuAPI we resolved the problem of reparsing and rewriting the same formula again and again. The next version of our Paracooba SAT and QBF solver already benefits

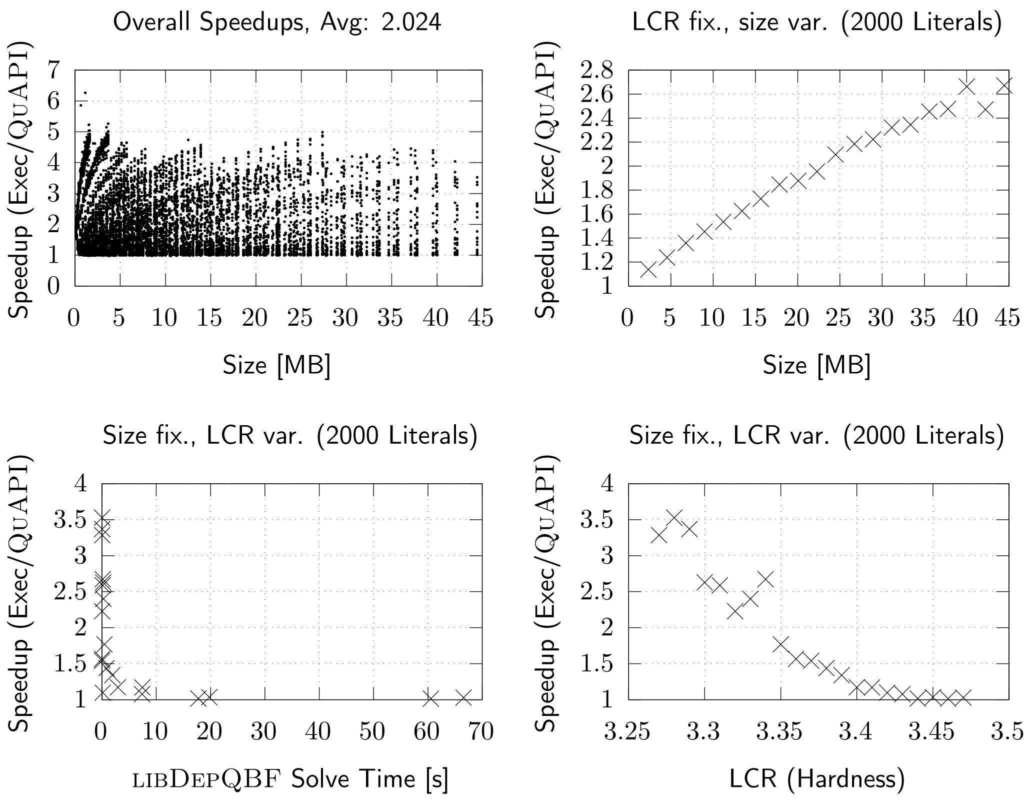

Figure 4: Speedup of using QuAPI compared to using the solver binary with files.

from this, enabling a portfolio of solvers that would each require specific integration code otherwise. Incremental solvers, however, also exploit inferred knowledge from previous runs that is still valid under the recent assumptions. At the moment, QuAPI always starts from scratch with each new set of assumption literals. To learn from previous runs, proofs produced by the solver could be analyzed and promising clauses (or cubes) could be provided to the solver. In addition to local solvers being executed with `fork`, another goal is to survey the potential of connecting to remote solvers using a TCP socket and delta-encoded transfers instead of local pipes. This would enable adding cloud-solvers retroactively into already existing applications to possibly improve run-times.

References

[1] A. Nadel, Boosting minimal unsatisfiable core extraction, in: R. Bloem, N. Sharygina (Eds.), Proc. of 10th Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), IEEE, 2010, pp. 221–229.

[2] C. Mencía, A. Previti, J. Marques-Silva, Literal-based MCS extraction, in: Proc. of

the 24th Int. Joint Conf. on Artificial Intelligence (IJCAI), AAAI Press, 2015, pp. 1973–1979.

[3] M. Heule, O. Kullmann, S. Wieringa, A. Biere, Cube and conquer: Guiding CDCL SAT solvers by lookaheads, in: Proc. of the 7th Haifa Verification Conference (HVC), Springer, 2011, pp. 50–65. doi:10.1007/978-3-642-34188-5_8.

[4] N. Eén, N. Sörensson, Temporal induction by incremental SAT solving, Electron. Notes Theor. Comput. Sci. 89 (2003) 543–560. doi:10.1016/S1571-0661(05)82542-3.

[5] N. Eén, N. Sörensson, An extensible sat-solver, in: Proc. of the 6th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 2919 of *Lecture Notes in Computer Science*, Springer, 2003, pp. 502–518. doi:10.1007/978-3-540-24605-3_37.

[6] A. Nadel, V. Ryvchin, Efficient SAT solving under assumptions, in: Proc. of the 15th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 7317 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 242–255. doi:10.1007/978-3-642-31612-8_19.

[7] A. Nadel, V. Ryvchin, O. Strichman, Preprocessing in incremental SAT, in: Proc. of the 15th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 7317 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 256–269. doi:10.1007/978-3-642-31612-8_20.

[8] A. Nadel, V. Ryvchin, O. Strichman, Ultimately incremental SAT, in: Proc. of the 17th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 8561 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 206–218. doi:10.1007/978-3-319-09284-3_16.

[9] R. Hickey, F. Bacchus, Speeding up assumption-based SAT, in: Proc. of the 22nd Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 11628 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 164–182. doi:10.1007/978-3-030-24258-9_11.

[10] C. Miller, P. Marin, B. Becker, Verification of partial designs using incremental QBF, AI Commun. 28 (2015) 283–307. doi:10.3233/AIC-140633.

[11] O. Beyersdorff, J. Mikolás, F. Lonsing, M. Seidl, Quantified Boolean formulas, in: Handbook of Satisfiability, volume 336, IOS Press, 2021, pp. 1177 – 1221. doi:10.3233/FAIA201015.

[12] L. Pulina, M. Seidl, The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17), Artif. Intell. 274 (2019) 224–248. doi:10.1016/j.artint.2019.04.002.

[13] F. Lonsing, U. Egly, Incremental QBF solving by DepQBF, in: Proc. of the 4th Int. Congress on Mathematical Software (ICMS), volume 8592 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 307–314. doi:10.1007/978-3-662-44199-2_48.

[14] M. Heisinger, M. Fleury, A. Biere, Distributed cube and conquer with paracooba, in: Proc. of the 23rd Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 12178 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 114–122. doi:10.1007/978-3-030-51825-7_9.

[15] M. N. Rabe, L. Tentrup, CAQE: A certifying QBF solver, in: Proc. of the Int. Conf. on Formal Methods in Computer-Aided Design (FMCAD), IEEE, 2015, pp. 136–143. doi:10.1109/FMCAD.2015.7542263.

[16] M. Janota, W. Klieber, J. Marques-Silva, E. M. Clarke, Solving QBF with

counterexample guided refinement, Artif. Intell. 234 (2016) 1–25. doi:10.1016/j.artint.2016.01.004.

[17] T. Balyo, A. Biere, M. Iser, C. Sinz, SAT race 2015, Artif. Intell. 241 (2016) 45–65. doi:10.1016/j.artint.2016.08.007.

[18] A. Ignatiev, A. Morgado, J. Marques-Silva, PySAT: A Python toolkit for prototyping with SAT oracles, in: Proc. of the 21st Int. Conf. on Theory and Applications of Satisfiability Testing (SAT), volume 10929 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 428–437. doi:10.1007/978-3-319-94144-8_26.

[19] fmtlib, {fmt}, https://github.com/fmtlib/fmt, 2022. Accessed 2022-02-24.