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

Maximilian Heisinger, Martina Seidl, Armin Biere

Institute for Symbolic AI, Johannes Kepler University Linz, Austria

2022-09-14

Some Preliminaries

Formulas are given in the (Q)DIMACS format and are in CNF.

Assumptions may be applied to a formula F by setting some variables in F to known values.

Solvers are any given executable that reads (Q)DIMACS from STDIN using standard (or wrapped) C functions like `getc`.

Some Preliminaries and a Spoiler

Formulas are given in the (Q)DIMACS format and are in CNF.

Assumptions may be applied to a formula F by setting some variables in F to known values.

Solvers are any given executable that reads (Q)DIMACS from STDIN using standard (or wrapped) C functions like `getc`.

QuAPI wraps *any given solver binary* and adds assumption-based reasoning support, *without* additional solver-specific efforts.

Motivating Example

`test.dimacs` contains a trivial formula with just one variable:

```
p cnf 1 1
1 0
```

```
quapify "test.dimacs"
```

Motivating Example

we add assumptions using `-a <int+>`

```
quapify "test.dimacs" -a 1 -a -1
```

Motivating Example

everything after `--` is the solver binary to be executed, including optional arguments

```
quapify "test.dimacs" -a 1 -a -1 -- caqe
```

Motivating Example

setting $x = \top$ lets the formula trivially evaluate to $\top$

```
quapify "test.dimacs" -a 1 -a -1 -- caqe
979423 0.000979 10 0
```

Motivating Example

setting $x = \top$ ($x = \bot$) lets the formula trivially evaluate to $\top$ ($\bot$)

```
quapify "test.dimacs" -a 1 -a -1 -- caqe
979423 0.000979 10 0
924825 0.000925 20 1
```

Motivating Example

applying -p also echoes out the applied assumption for each result

```
quapify "test.dimacs" -a 1 -a -1 -p -- caqe
979423 0.000979 10 0 1
924825 0.000925 20 1 -1
```

What did we just see?

- We took the QBF solver *caqe* (written in Rust) as a binary executable,

What did we just see?

- We took the QBF solver *caqe* (written in Rust) as a binary executable,
- gave it a formula in DIMACS format,

What did we just see?

- We took the QBF solver *caqe* (written in Rust) as a binary executable,
- gave it a formula in DIMACS format,
- and solved it under two different assumptions,

What did we just see?

- We took the QBF solver *caqe* (written in Rust) as a binary executable,
- gave it a formula in DIMACS format,
- and solved it under two different assumptions,
- *without executing the binary multiple times!*

What did we just see?

- We took the QBF solver *caqe* (written in Rust) as a binary executable,
- gave it a formula in DIMACS format,
- and solved it under two different assumptions,
- *without executing the binary multiple times!*

$\Rightarrow$ generic assumption-based reasoning, interactively or as a library

Just use a solver's API!

1. Some solvers, especially in QBF (e.g. *caqe* or *Rareqs*), do not have APIs.

Just use a solver's API!

1. Some solvers, especially in QBF (e.g. *caqe* or *Rareqs*), do not have APIs.
2. Different solving paradigms and algorithms may complement each other.

Just use a solver's API!

1. Some solvers, especially in QBF (e.g. *caqe* or *Rareqs*), do not have APIs.
2. Different solving paradigms and algorithms may complement each other.
3. No unifying interface (IPASIR doesn't implement unified dispatching).

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed vars n */
                             NULL, /* SAT regex */
                             NULL  /* UNSAT regex */);
```

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed vars n */
                             NULL, /* SAT regex */
                             NULL  /* UNSAT regex */);
quapi_quantify(s, -1); /* ∀x1 */
quapi_quantify(s, 2); /* ∃x2 */
```

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed 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 */
```

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed 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 */
```

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed 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! */
```

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed 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! */
```

Assumption-Based Reasoning as a Library

```
quapi_solver *s = quapi_init("caqe", /* solver path */
                             NULL, /* argv */
                             NULL, /* envp */
                             2,    /* variables */
                             2,    /* clauses */
                             1,    /* number of assumed 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 */
```

The Two Pillars of QuAPI

"Combining `LD_PRELOAD` hacking with `fork`"

The Two Pillars of QuAPI

"Combining `LD_PRELOAD` hacking with `fork`"

LD_PRELOAD

- Environment variable
- Tells the linker `ld` to load symbols from a shared object (`.so`) before other ones
- Makes overriding symbols possible

The Two Pillars of QuAPI

"Combining `LD_PRELOAD` hacking with `fork`"

`LD_PRELOAD`

- Environment variable
- Tells the linker `ld` to load symbols from a shared object (`.so`) before other ones
- Makes overriding symbols possible

`fork`

- Copies the current process into two
- Copies (copy-on-writes) a process' memory and whole runtime state as-well
- Keeps file descriptors open (unless opted out)

QuAPI Control Flow

1. Override a solver's `read` with the one provided by QuAPI using `LD_PRELOAD`,

QuAPI Control Flow

1. Override a solver's `read` with the one provided by QuAPI using `LD_PRELOAD`,
2. feed it with re-stringified (Q)DIMACS,

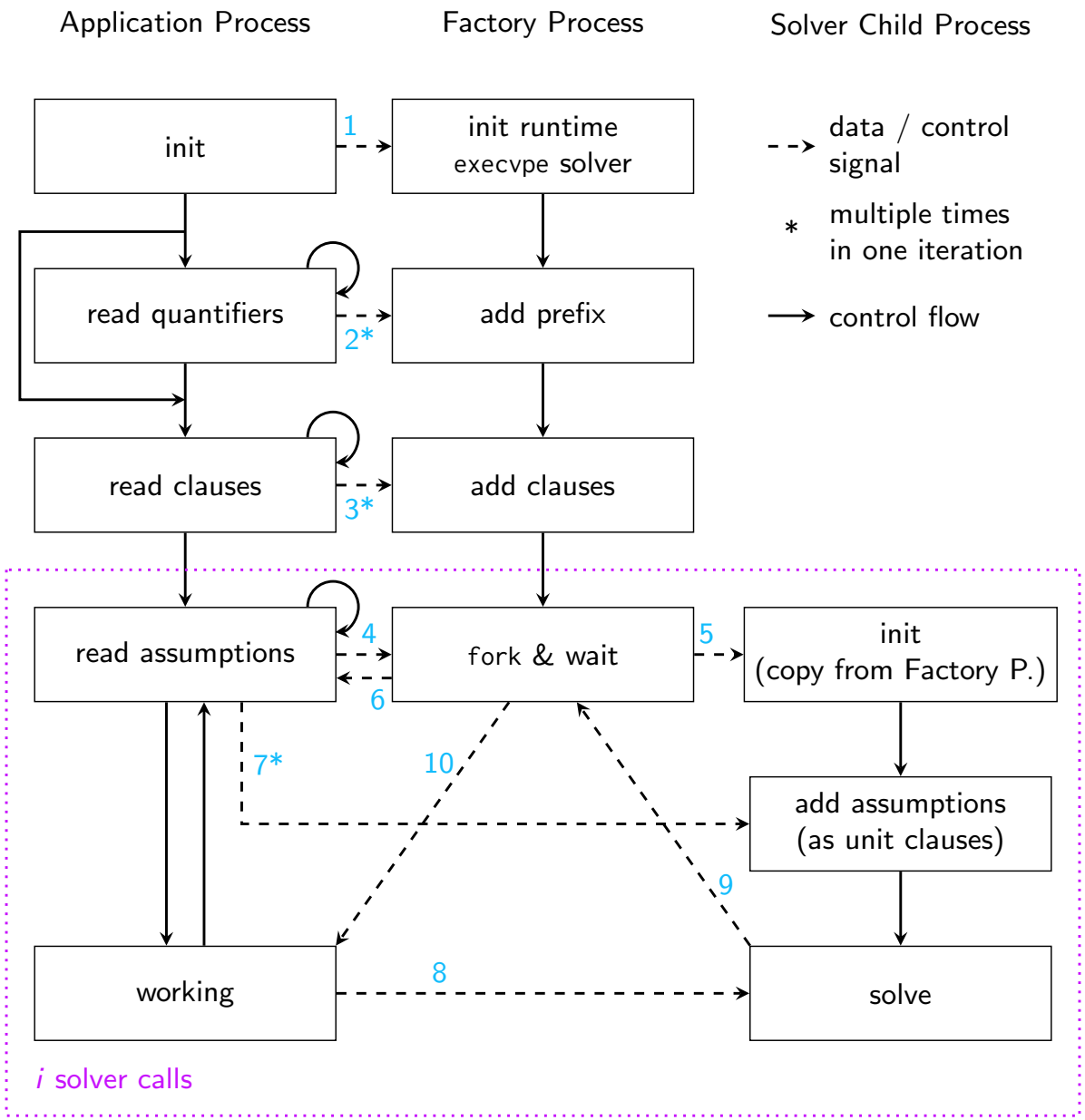
QuAPI Control Flow

1. Override a solver's `read` with the one provided by QuAPI using `LD_PRELOAD`,
2. feed it with re-stringified (Q)DIMACS,
3. `fork` the fully initialized solver factory process once for every time assumptions are applied,

QuAPI Control Flow

1. Override a solver's `read` with the one provided by QuAPI using `LD_PRELOAD`,
2. feed it with re-stringified (Q)DIMACS,
3. `fork` the fully initialized solver factory process once for every time assumptions are applied,
4. and return the result to the grandparent process.

QuAPI Control Flow in More Detail



About Speed

- Faster than calling solver multiple times

About Speed

- Faster than calling solver multiple times
- Even when a solver is just reading from a RAM-disk.

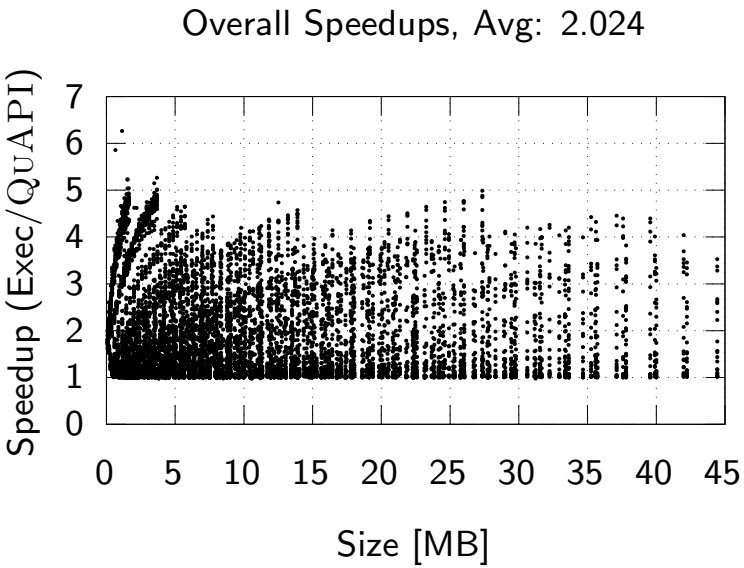
About Speed

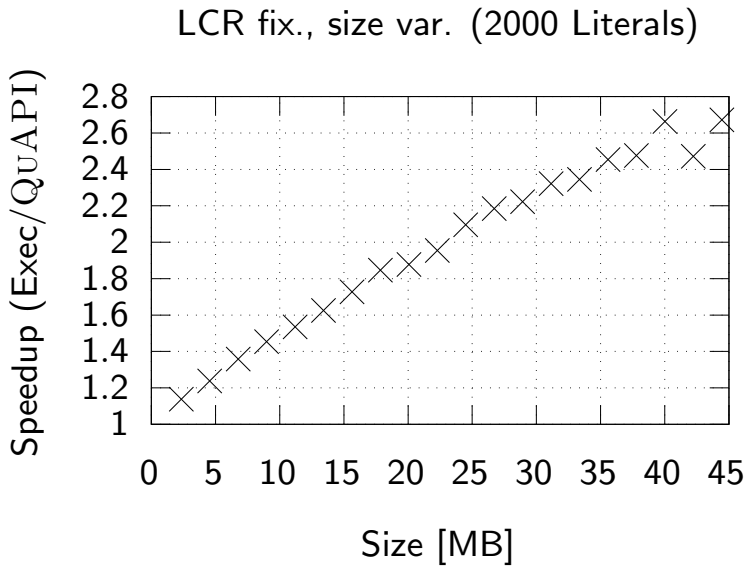
- Faster than calling solver multiple times
- Even when a solver is just reading from a RAM-disk.
- Can be faster for even just one assumption (more efficient parsing, inter-process-communication and stringification).

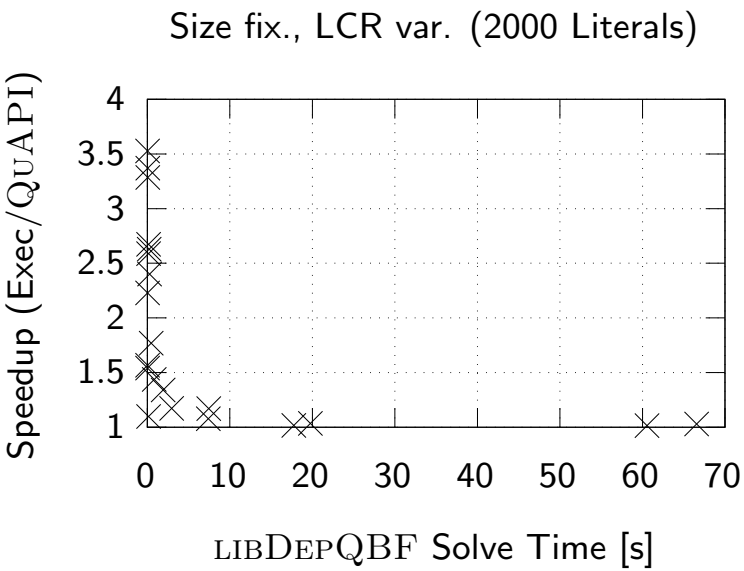
About Speed

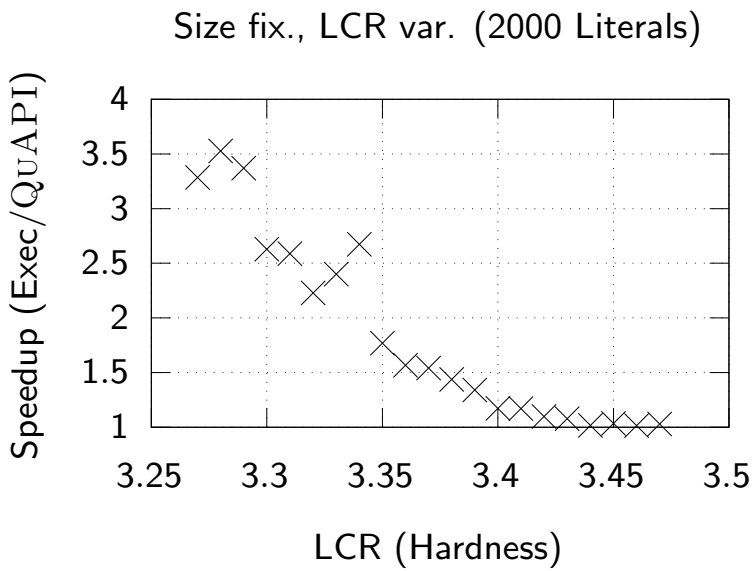
- Faster than calling solver multiple times
- Even when a solver is just reading from a RAM-disk.
- Can be faster for even just one assumption (more efficient parsing, inter-process-communication and stringification).
- More speedup the faster each assumption is solved (i.e. the easier the problem), the larger the problems are, and the more assumptions are applied.

Speedup Summary Plot







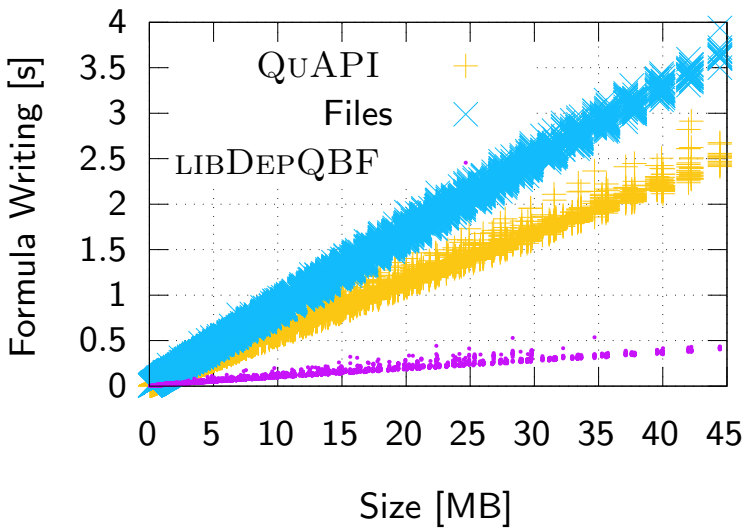


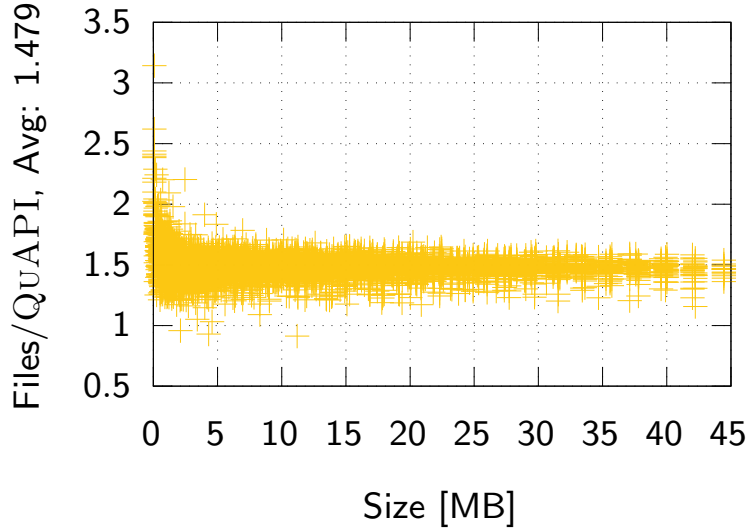
Using QuAPI in Your Projects

Available under MIT license at:

github.com/maximaximal/QuAPI

Overhead in Writing Formulas





Overhead in Applying Assumptions