Optimizing the COHO Reachability System: A Practical Handbook

Paul Liu

Abstract

We perform an in-depth profiling of the COHO reachability tool on several test problems, with the goal of optimizing COHO to be competitive with other reachability tools. Our optimizations impact COHO from end to end, ranging from IO optimizations on the Matlab side to performance improvements on the Java side. Our main results are: (1) a factor of 2.8 speed up on non-linear systems on the Java back end, (2) a factor of 1.2 speed up when using COHO through the Matlab interface, and (3) a new pivoting rule to COHO's linear program solver that reduces the number of pivots by a factor of 1.6. Furthermore, we optimize and test a parallel version of COHO for linear systems that shows a speed up factor of over 20 when tested on a high dimensional linear system. Finally, we compare the optimized COHO's performance to the SpaceEx reachability package. Though we are slower than SpaceEx by a factor of 2, we show that with future work and specific optimizations beyond the scope of this project, COHO's performance can surpass that of SpaceEx.

We intend for this project to not only be a list of the optimizations made to COHO, but also to be of future use as an informal road map of potential optimizations and common traps one can fall into when optimizing COHO.

1 Introduction

Due to the rapidly increasing complexity of hardware and software, the effectiveness of traditional verification techniques is quickly waning. This has led increased interest in recent years in automated formal verification techniques. In many systems, such as those of analog circuits, modeling methods must model both the discrete and continuous nature of the system, resulting in a hybrid system. In such models, we are often interested in the reachable domain of a set of initial conditions, to ensure that no initial condition leads to an unwanted state. COHO is a reachability analysis tool that solves this problem for continuous and hybrid systems, developed originally for verification of analog or mixed circuits [11]. Its key novelty is that it uses a projection based technique to efficiently represent high dimensional problems as a set of projections in lower dimensions. COHO has had success in verifying a variety of hybrid systems, and has proven to be an effective tool [18]. However, some initial testing suggests that the performance of COHO could be optimized, especially with regards to linear hybrid systems.

In this project, we attempt to optimize the COHO reachability analysis tool. Our optimizations impact COHO from end to end, ranging from IO optimizations on the Matlab side to performance improvements on the Java side. Our ultimate goal is to make the speed of COHO comparable to other reachability tools, such as SpaceEx [7].

We begin with some necessary background on COHO in Section 2, describe our profiling methodology in Section 3, our optimizations in Sections 4 to 6, and identify areas worthy of future optimization in Section 7. We intend for this project to not only showcase the optimizations made to COHO, but also to be of use as an informal road map of future optimizations and common traps one can fall into when optimizing COHO. Hence, we try to not list just the optimizations themselves, but also interesting roadblocks encountered along the way.

2 Background

In a typical reachability problem. we are given a region $\Omega \in \mathbb{R}^n$ and a model F of some system, usually represented as an ordinary differential inclusion $\dot{x}(t) \in F(x, t)$. Given Ω and F, the reachable space at time t is the space

$$\mathcal{S}_t = \{x(t) \,:\, x(0) \in \Omega \wedge \dot{x} \in F\}.$$

In other words, $\mathcal{S}_t$ is the set of all possible positions that we can reach when our initial state is in Ω. Typically, we are interested in whether $\mathcal{S}_t$ satisfies some set of constraints for all t, such as the constraint that $\mathcal{S}_\infty$ converges to a single point or is bounded. A straightforward approach to estimating $\mathcal{S}_t$ is to simply sample points from Ω and simulate a large set of initial conditions. While practical, this approach is deficient as it does not prove anything about the properties of $\mathcal{S}_t$ for unsampled initial conditions. For verification problems, we seek a method which provides an upper bound $\mathcal{S}'_t$ such that $\mathcal{S}_t \subseteq \mathcal{S}'_t$ for all times t. Checking the given constraints on $\mathcal{S}'_t$ gives us the property that incorrect systems are never falsely verified, although the over-approximation might make it fail to verify a correct system.

This is the main purpose of COHO, a tool for reachability analysis and verification. The novelty of COHO is that although high dimensional problems are hard to deal with, two-dimensional problems are relatively easier. As such, COHO represents reachable regions by their projection onto two-dimensional subspaces. For each projection in 2D, COHO computes (and over-approximates) their time evolution, and then reassembles the projections in the original space in such a way that the over-approximation is preserved.

To compute the evolution of reachable states, COHO solves a large number of linear programs. More details on the internals of COHO can be found in Greenstreet and Mitchell's original papers [10, 11]. We mention that COHO is not the only tool for reachability analysis. A comprehensive listing of other tools, can be found in Yan's PHD thesis [19].

To optimize COHO, it is necessary to understand the its pipeline as well as how to use it. The bulk of COHO is written in Java, which forms the backend of projection operations and linear programming. The front end is written in Matlab, and handles parts of the projection operation, as well as discretizing the reachability computation across small time steps. To use COHO, one can use the Java side directly by specifying a log file to read problems from, or more commonly, use the Matlab interface. To send a problem to the Java backend, the Matlab interface simply writes to a pipe that a Java thread constantly reads from. Once the backend reads a Matlab request from the pipe, it computes a solution and then writes it to another pipe that the Matlab interface reads from. The communication process is thus simple and easy to debug. However, it has its issues, as we'll see in Section 5.

When the Java backend reads in a problem, there are two main stages that COHO goes through. In the first stage, COHO projects the problem down into 2D and makes over-approximations as necessary. This involves common geometric algorithms such as computing convex hulls as well as possibly reducing and pruning the number of edges in the projection polygon. In the second stage, linear programs are constructed from the given problem and simplex is used to solve it. Due to the special form of the COHO linear program, several algorithmic optimizations differentiates COHO's simplex routine from a standard simplex. We will not go over the details here, but instead refer the reader to Laza's work [13, 21]. Typically, COHO's computations are done in double precision, with interval bounds on the errors of all computations involved. To ensure that results produced are accurate and that no significant loss of precision occurs, COHO repeats all computations in arbitrary precision rationals whenever the error interval grows too large. Thus COHO's results are always guaranteed to be accurate. However, as we'll see in Section 4, the usage of arbitrary precision rationals has its downsides, the most notable of which is that computations in arbitrary precision are inexorably slow.

3 Methodology

To optimize COHO we first separated the optimization of its backend and frontend. To measure only the backend, we profiled the backend performance on the logs file of typical problems sent to COHO. Since the Matlab interface sends all its requests through a pipe, we simply copied all information sent through the pipe into a log. The logs were taken from some examples previously tested in COHO [19], as well as a high dimensional problem from SpaceEx [7]. This allows us to effectively remove communication delays between COHO and Matlab, and focus solely on bottlenecks present in COHO's Java side. A full list of problems tested is given in Table 1.

To profile operations on the Java backend, we used the VisualVM Java profiler [1].

Table 1: Example problems used in Coho. The bottom half was only used for verification at the end of the optimization process.

Problem	Dim	Note
ex_28heli	28	SpaceEx helicopter controller example
ex_3vdp	3	Three dimensional Van der Pol oscillator
ex_3dm	3	Dang and Maler's example
ex_2vdp	2	Two dimensional Van der Pol oscillator
ex_2sink	3	Two dimensional sink example
ex_3pd	3	Play-Doh example

Though there are a variety of other profilers, we chose VisualVM as it is fast enough to have a negligible effect on the running time of the code, and robust enough to avoid the the effects of "profiler bias", a scenario in which simply running a profiler on a program alters its efficiency.

Instead of measuring purely the frontend Matlab performance of COHO, we measured its end to end performance (i.e. including the Java backend). This allowed us to profile communication inefficiencies between the frontend and backend, in addition to determining the slowest parts of COHO as a whole. To measure the end to end performance, we used Matlab's built-in profiler on the same problems listed in Table 1. However, this presented some challenges (e.g. profiler bias) as we'll see in Section 5.

Finally, we needed to verify that none of our optimizations to COHO resulted in incorrect results. This was simple to address: for each of the log files sent to COHO, there was a resulting log file sent back from COHO to Matlab. By ensuring that these log files remained the same throughout all of our optimizations, we were confident that our optimizations were correct.

Though it proved to be ultimately unnecessary, we also developed our Java optimizations on only the top half the log files in Table 1, and tested the performance on the other half of the log files only after all optimizations were complete, to ensure that our optimizations were sufficiently general, and not specifically tuned to solve only the problems we were testing on. The Matlab optimizations were tested on all the log files, as only IO optimizations were made.

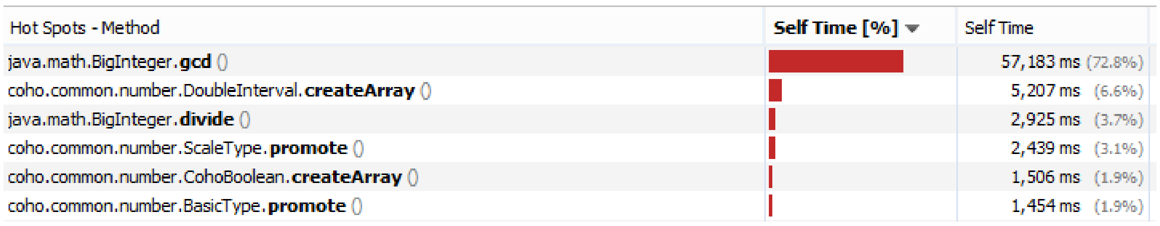
All testing was done on **newcastle.cs.ubc.ca**, a Linux machine with 8 Intel Xeon E7 CPUs, and 132 GB of ram.

4 COHO Java backend Improvements

4.1 Improvements to CohoAPR

We begin with a profile of COHO on a typical log file (Figure 1). As the profile clearly shows, the majority (roughly 70%) of COHO's time is spent computing with COHO's arbitrary precision rationals (CohoAPR), specifically in the BigInteger.gcd operation. Digging further, all calls to BigInteger.gcd occurs in CohoAPR's normalize function

Figure 1: Profile of Coho's backend on ex_3vdp. Shows the top 6 most time consuming functions. We do not show the profile results for the other examples as they look similar across all 3 example problems.



Hot Spots - Method	Self Time [%]	Self Time
java.math.BigInteger.**gcd** ()		57,183 ms (72.8%)
coho.common.number.DoubleInterval.**createArray** ()		5,207 ms (6.6%)
java.math.BigInteger.**divide** ()		2,925 ms (3.7%)
coho.common.number.ScaleType.**promote** ()		2,439 ms (3.1%)
coho.common.number.CohoBoolean.**createArray** ()		1,506 ms (1.9%)
coho.common.number.BasicType.**promote** ()		1,454 ms (1.9%)

(`coho.common.numbers.CohoAPR.normalize`). The normalize function is called on every CohoAPR operation, to ensure that the APR is in reduced form and to mitigate the problem of the numerator and denominator size increasing exponentially after every APR operation. Though the purpose of normalize is well intended, it actually hurts the performance of COHO, as the linear systems COHO deals with are usually small in size, and so the precision needed for APR rationals is typically not large. Furthermore, it is not necessary to call normalize after every COHO operation, as operations like negation and taking reciprocals of a number clearly preserve the greatest common divisor of the numerator and denominator.

Thus, our first optimizations were to simply reduce the number of calls to normalize by removing it where it is not needed, and implement a fast binary GCD algorithm [5] in the case where the numerator and denominator were small enough to be represented by 64-bit integers. We expected this to improve COHO's performance substantially. Much to our surprise, the changes resulted in a very modest improvement to COHO's performance, much less than what we were expecting.

A subtle bug in COHO fraction. As stated, the linear systems COHO deals with are typically small in size, so we expected the precision needed for APR rationals to typically be small as well. To confirm this, we recorded the number of bits needed to represent the inputs to every normalize call (Figure 2). Contrary to our expectations, most of the inputs required a large number of bits to represent. Strangely though, a large number of inputs required exactly 0 bits for the numerator, and exactly 1024 bits for the denominator. In other words, the numerator was 0, so the fraction should have been 0/1. However, it was represented as $0/X$, where X needed 1024 bits to be represented. Further testing revealed that $X = 2^{1023}$ in every such call to normalize. As it turns out, the cause of this was in a function completely unrelated to normalize. When COHO switches from 64-bit doubles to arbitrary precision, all existing 64-bit double values are converted to CohoAPRs via IEEE 754 specifications [2]. In other words, CohoAPR represents the given double exactly as $x/2^y$, where x is the significand and y is the exponent subtracted by the bias (which is fixed to be 1023). Due to the way doubles are represented, x and y are coprime since x is guaranteed to have no factor of 2, so the numerator and denominator is set to x and 2^y respectively without additional checks. The only exception

Figure 2: Histograms of input bit length in CohoAPR.normalize.

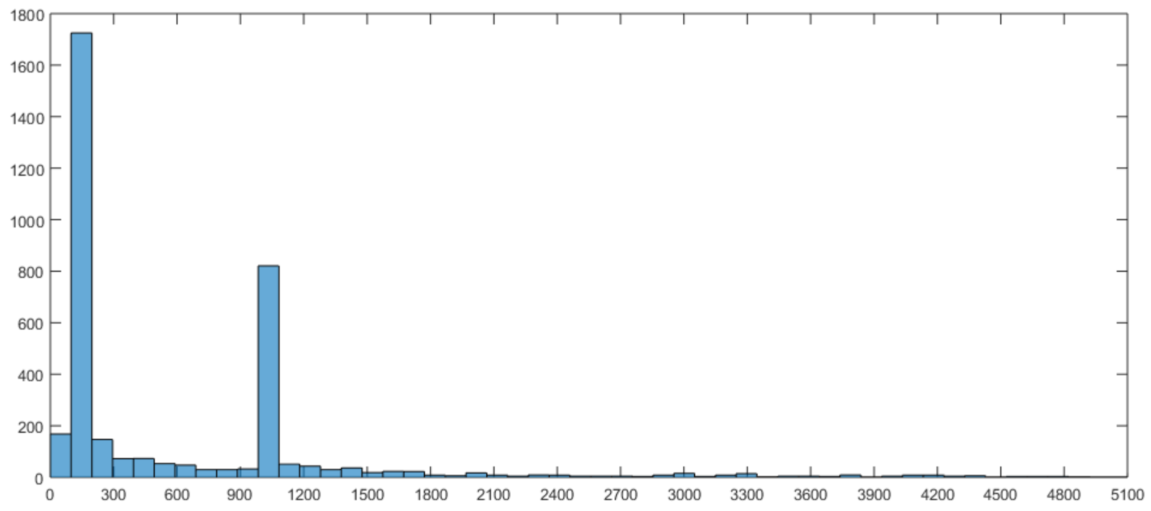
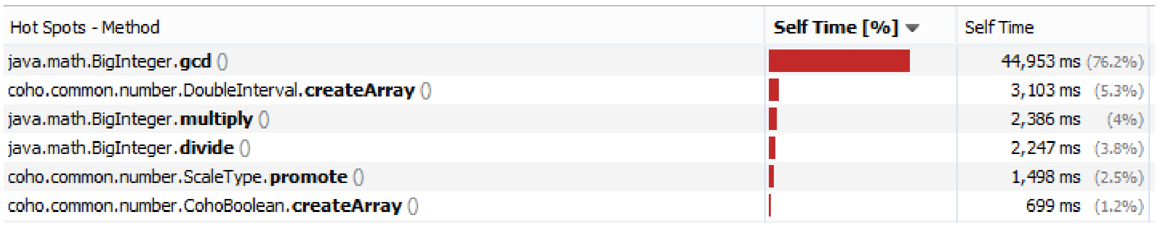


Figure 3: Profile of Coho's backend on ex_3vdp after fixing the double conversion bug.



Hot Spots - Method	Self Time [%] ▼	Self Time
java.math.BigInteger.**gcd** ()	🟥	44,953 ms (76.2%)
coho.common.number.DoubleInterval.**createArray** ()	🟥	3,103 ms (5.3%)
java.math.BigInteger.**multiply** ()	🟥	2,386 ms (4%)
java.math.BigInteger.**divide** ()	🟥	2,247 ms (3.8%)
coho.common.number.ScaleType.**promote** ()	🟥	1,498 ms (2.5%)
coho.common.number.CohoBoolean.**createArray** ()	🟥	699 ms (1.2%)

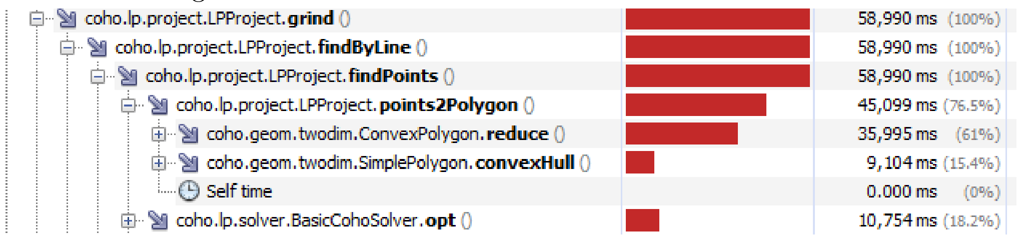
to this case is when the double is exactly 0, causing COHO to misconvert a double value of 0 to $0/2^{1023}$.

After fixing this bug, COHO's the previous optimizations improved COHO's performance noticeably. Our final optimization was to only run the BigInteger GCD algorithm when the number of bits needed to represent the CohoAPR exceeded 5000 (experimentally determined to be a good threshold). The profile after our optimizations are shown in Figure 3. Note that the time of GCD operations reduced by 25% without increasing the time significantly for any other operations.

4.2 Improvements to geometric subroutines

Recall that COHO's operations proceeds in two stages, a geometric projection stage and a linear solver stage. Although optimizing CohoAPR is a good start, the best optimization is to reduce the usage of APRs until it is negligible. In this section, we reduce usage of CohoAPR this by optimizing the geometry stage of COHO. As we can see from Figure 4, the a large portion (about 15%) of COHO's time is oddly spent within the convex hull algorithm. Furthermore, a whopping 61% is spent on COHO's reduce function. Overall,

Figure 4: Profile of Coho's call tree after Section 4.1.



```
coho.lp.project.LPProject.grind ()                           58,990 ms  (100%)
  coho.lp.project.LPProject.findByLine ()                    58,990 ms  (100%)
    coho.lp.project.LPProject.findPoints ()                  58,990 ms  (100%)
      coho.lp.project.LPProject.points2Polygon ()            45,099 ms (76.5%)
        coho.geom.twodim.ConvexPolygon.reduce ()             35,995 ms   (61%)
        coho.geom.twodim.SimplePolygon.convexHull ()          9,104 ms (15.4%)
        Self time                                             0.000 ms    (0%)
      coho.lp.solver.BasicCohoSolver.opt ()                  10,754 ms (18.2%)
```

these two functions account for 75% of COHO's total running time, which is far larger than the time taken by simplex (shown as BasicCohoSolver.opt in Figure 4).

4.2.1 Optimizing COHO's convex hull routine

Digging further, we discovered that the the majority of the convex hull time was spent in a single call to Java's built-in sort. The sort was required as a preprocessing stage to the convex hull algorithm, and required a special angle comparator. Each call to the comparator required up to 6 arithmetic CohoAPR operations, in order to compare the orientation angle between two given line segments. Thus, our first optimization was to remove the comparator if possible.

Prior to our optimizations, COHO used the Graham scan algorithm [5] for computing a convex hull. To improve performance, we switched to Andrew's monotone chains algorithm [3]. Although their asymptotic complexities are the same, Andrew's algorithm turns out to be much more efficient when dealing with APRs. This is due to the fact that Andrew's monotone chains does not require an angle sort at all. Instead, it sorts by y-coordinate, and then proceeds the exact same way as Graham scan. Sorting the points by their y-coordinates allows us to use a much faster comparator, and avoid the costly angle comparator entirely.

In order to make the convex hull time truly negligible, we further noted that all of COHO's APR operations in the convex hull algorithm resulted from COHO's baseTurn routine. Given three points (a_x, a_y), (b_x, b_y), and (c_x, c_y), the routine determines if three given points form a right hand turn or not by checking if

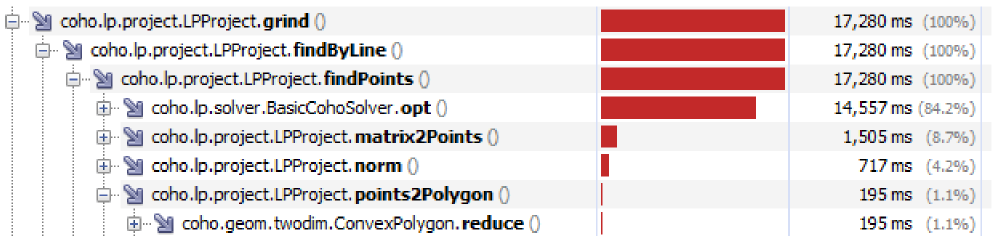
$$a_x b_y + b_x c_y + c_x a_y - a_x c_y - b_x a_y - c_x b_y > 0. \tag{1}$$

To improve the speed of this routine, we noted that the original expression for computing a right hand turn required computing 4 additions and 6 multiplies (see Equation 1). However, we were able to factor this expression into one with 4 additions and just 2 multiplies by noting that

$$a_x b_y + b_x c_y + c_x a_y - a_x c_y - b_x a_y - c_x b_y = (a_x - c_x) \cdot (b_y - c_y) - (b_x - c_x) \cdot (a_y - c_y). \tag{2}$$

Additional experiments showed that in most cases, the values of det1 and det2 quite a bit in magnitude. Since baseTurn only requires knowing which of det1 and det2 are

Figure 5: Profile of Coho's call tree after Section 4.1.



```
coho.lp.project.LPProject.grind ()                          17,280 ms  (100%)
  coho.lp.project.LPProject.findByLine ()                   17,280 ms  (100%)
    coho.lp.project.LPProject.findPoints ()                 17,280 ms  (100%)
      coho.lp.solver.BasicCohoSolver.opt ()                 14,557 ms  (84.2%)
      coho.lp.project.LPProject.matrix2Points ()             1,505 ms  (8.7%)
      coho.lp.project.LPProject.norm ()                        717 ms  (4.2%)
      coho.lp.project.LPProject.points2Polygon ()              195 ms  (1.1%)
        coho.geom.twodim.ConvexPolygon.reduce ()               195 ms  (1.1%)
```

larger, our last optimization was to first try to compute Equation 2 in double precision, and then in APR if the precision is not enough. Implementing these operations improved the performance of COHO's convex hull by at least a factor of 100, to the point where the profiler cannot even pick up the convex hull execution time (i.e. it is less than 100ms). This also significantly decreased the runtime of COHO's reduce function, as much of the time within the function was spent evaluating Equation 1.

4.2.2 Optimizing COHO's reduce edges routine

As discussed in Section 2, the reduce function is used for pruning the projections in the case that they have too many edges or points. To evaluate which edge and points to prune, COHO uses a heuristic scoring formula, where objects with smaller score are more likely to be pruned. The most computationally intensive part of this heuristic requires COHO to evaluate the area of a simple polygon, which takes a prohibitively long time when computing with APRs. Our experiments show that among roughly equal scoring edges, choosing one with slightly lower score does not noticeably affect the accuracy of COHO. In other words, it is fine in most cases to evaluate the area of the polygon with doubles, and then fall back to APR if scores are close. This improved the performance of the reduce function dramatically. Following these optimizations, the runtime of reduce decreased by a factor of at roughly 180.

A profile after all optimizations of Section 4.1 and 4.2 are made can be found in Figure 5.

4.3 Improvements to COHO's simplex solver

Our final optimization is a small improvement to COHO's simplex solver. After our optimizations in the previous sections, the majority of the time is spent in BasicCohoSolver.opt. However, much of the time is not spent on simplex, but instead on the final step of simplex, where we check if a basis is feasible by checking dual feasibility. To be careful, this is done in CohoAPR. In fact, time spent in simplex does not seem to be an issue in our test examples, as in none of our test problems did the simplex solver switch to APR mode (or if it did, it was not significant enough to be visible on the profiler). However, we recognize that for future problems, the bottleneck could easily be the large number of pivots made in simplex when it is in APR mode. Thus, for preventive measures, we changed to a more efficient pivoting rule in COHO's simplex routine.

As noted by Yan [17], COHO sometimes performs a large amount pivots for its simplex algorithm. Prior to our optimizations, COHO was using Bland's rule to avoid cycling [4]. Now it uses Dantzig's rule for pivoting [6]. Although Dantzig's rule may cycle, this never happens for COHO's test problems. Furthermore, Dantzig's rule can be made to be non-cycling [8], and it is a simple modification suitable for future work. Changing COHO's pivoting rule decreases the number of pivots by up to a factor roughly 2 in some test problems (Table 2), although this had no noticeable impact on the running time.

Table 2: Number of pivots for example problems.

Problem	Pivots in total (before)	Pivots in total (after)
ex_28heli	301	301
ex_3vdp	73582	43739
ex_3dm	18183	11392

4.4 Results of optimizations

The results after all optimizations of Sections 4.1-4.3 can be found in Table 3. As one can see, the speed across all problems increased by at least a factor of 2, with some problems having a factor of 3 increase.

Table 3: Execution times of the Java backend on all example problems of Table 1. The time measured is the wall clock time.

Problem	Time before (s)	Time after (s)	Time before / Time after
ex_28heli	290.9	140.1	2.1
ex_3vdp	68.0	32.0	2.1
ex_3dm	28.6	10.4	2.8
ex_2vdp	17.9	8.4	2.1
ex_2sink	7.2	3.0	2.4
ex_3pd	504.5	256.5	2.0

5 End to end optimization of Coho

Although the Java backend of COHO is fast, the Matlab side still leaves something to be desired. As seen in Table 4, running the example problems through the Matlab interface increases the time by roughly a factor of 2-10. To determine the cause of this, we use Matlab's built-in profiler on the **ex_2vdp** problem, as it experienced the largest slow-down.

Table 4: Execution times of COHO through the Matlab interface. The time measured is the wall clock time.

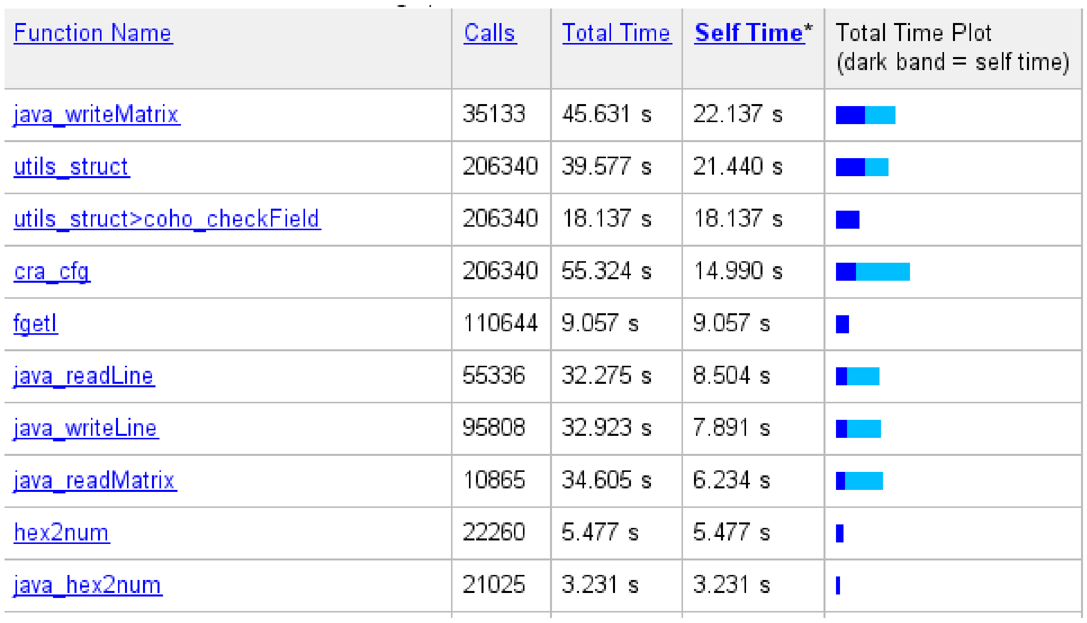
Problem	Time (s)	End to end time / Backend time
ex_28heli	217.2	1.6
ex_3vdp	132.9	4.2
ex_3dm	33.4	3.2
ex_2vdp	89.4	10.6
ex_2sink	19.8	6.7
ex_3pd	1390.1	5.2

Challenges to using Matlab's built-in profiler. Although the profiler within Matlab was mostly accurate, there were some unique challenges encountered along the way. Firstly, because the Matlab communications reads from a pipe, we do not know if the input reading is slow or if Matlab is simply hanging because the Java backend isn't computing fast enough. Secondly, Matlab runs on Java, and therefore uses a JIT to optimize its programs. When the profiler is turned on, many JIT optimizations are turned off [14], which may flip the order of whats fast and slow in the profile. Regardless, the profiler is still a useful tool for finding potential bottlenecks, simply because it is well written and also because there are no alternatives. Furthermore, all bottlenecks will still appear in the profile listing, although some may not be actual bottlenecks. One such example is the time spent in utils_struct and cra_cfg in Figure 6. Although the Matlab profiler lists this as one of the top bottlenecks, it is because utils_struct makes heavy use of structs and the JIT optimizations disabled by Matlab are exactly responsible for optimizing the performance of structs.

Disregarding the profile data of utils_strut and cra_cfg, it is clear from the profile in Figure 6, that the a major bottleneck is the inefficiency of the Matlab to Java communication process. A large amount of time is spent on IO functions, such as java_writeMatrix, java_readMatrix, and fgetl. Though these functions could have been slowed by missing JIT optimizations, we confirmed that the timings profiled by the profiler was accurate by timing those methods alone in reruns of the example without a profiler. Naturally, the optimal solution would be to rewrite the Matlab interface and send the problems from Matlab to COHO directly, instead of reading and writing to pipes. However, this was beyond the scope of our project due to time limitations. Hence we limited our changes to low level optimizations that would not affect the architecture of COHO.

Despite our efforts, we could not improve the write times of the Matlab interface as it was already using the lowest level of file write methods available in Matlab. However, we were able to improve the reading times of the Matlab interface, by writing performance critical functions such as java_hex2num and parts of java_readMatrix as a C++ MEX file for Matlab. Our results are shown in Table 5.

Figure 6: Output of Matlab profiler on ex_2vdp.



Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
java_writeMatrix	35133	45.631 s	22.137 s	▮
utils_struct	206340	39.577 s	21.440 s	▮
utils_struct>coho_checkField	206340	18.137 s	18.137 s	▮
cra_cfg	206340	55.324 s	14.990 s	▮
fgetl	110644	9.057 s	9.057 s	▮
java_readLine	55336	32.275 s	8.504 s	▮
java_writeLine	95808	32.923 s	7.891 s	▮
java_readMatrix	10865	34.605 s	6.234 s	▮
hex2num	22260	5.477 s	5.477 s	▮
java_hex2num	21025	3.231 s	3.231 s	▮

Table 5: Execution times of the Java backend on all example problems of Table 1. The time measured is the wall clock time.

Problem	Time before (s)	Time after (s)	Time before / Time after
ex_28heli	217.2	160.7	1.4
ex_3vdp	132.9	118.3	1.1
ex_3dm	33.4	30.1	1.1
ex_2vdp	89.4	74.4	1.2
ex_2sink	19.8	17.1	1.2
ex_3pd	1390.1	1276.1	1.1

6 Comparison with SpaceEx and Parallel Coho

As previously stated, our ultimate goal is to make COHO comparable with other reachability tools, in particular SpaceEx. To make our comparison, we tested the performance of COHO and SpaceEx on the SpaceEx helicopter problem (Table 6). As one can see, the performance gap is staggering: COHO takes roughly 16x the time SpaceEx does. This is not surprising, as SpaceEx is designed for linear hybrid automata, whereas COHO is general enough to handle non-linear models.

Prior to our optimizations, Yan implemented a parallel version of COHO [20] aimed at addressing the issue of the Java backend slowing down the COHO frontend for linear models. In Yan's parallel version, multiple threads with the COHO backend is spawned, all with their individual pipes for Matlab to read from. Hence the Matlab interface writes

Table 6: Execution times of COHO on the SpaceEx helicopter problem. The time measured is the wall clock time.

Problem	End to end COHO time (s)	Java backend time (s)	SpaceEx time (s)
ex_28heli	161.2	140.1	9.7

multiple problems to COHO's backend at once, and then receives multiple results in one sweep. This is only possible for linear models as the computations at each time step are decoupled from previous time steps, where as for non-linear model COHO requires the result of the previous time step to compute the current one. The performance of COHO is increased dramatically with this optimization, as each COHO thread receives much less work (Table 7). Overall, the Java backend runs over 20 times faster than the unoptimized COHO.

SpaceEx on the other hand, does not seem to exploit any sort of parallelism in their code. The same optimizations to the sequential COHO version apply to the parallel version, as the frontend and backend codes are essentially unchanged. Even with our optimizations to however, there is still a factor of 2 difference between between the end to end performance of COHO and SpaceEx. There is a silver lining however, as we note that for machines with more cores, the performance of COHO will only grow faster, whereas the performance of SpaceEx stagnates. Hence simply testing COHO on a machine with more cores will make COHO faster than SpaceEx.

Table 7: Execution times of Parallel COHO on the SpaceEx helicopter problem. The time measured is the wall clock time.

Problem	End to end COHO time (s)	Java backend time (s)	SpaceEx time (s)
ex_28heli	21.3	14.1	9.7

7 Road map for Future Optimizations

Although many optimizations were found for COHO, there remains many more potential optimizations to be done.

In Section 4, we noted that the majority of the time went to computing with CohoAPR. Despite our optimizations, this remains the largest overhead. However, we note that CohoAPR is built off Java's BigInteger library, which is known to be quite inefficient. In particular, GNU's GMP library is order of magnitudes faster, and has support for arbitrary precision rationals [9]. Incorporating this into COHO will likely increase the speed greatly. However, it is no easy task as all of GMP's operations are *in-place*, so that object creation is avoided as much as possible. This is at odds to COHO's current architecture, where a new object is created for the result of each computation. Incorporating GMP thus remains an avenue for future exploration.

In the geometry portion of Section 4, we optimized away the cost of computing the convex hull and reduce. However, some portion of time is still spent on the reduce function, where APR is used for segment-segment intersections. One potentially effective optimization to try is to simply compute the intersection point in doubles. However, one has to be careful in that no under-approximations occurs if this intersection point is added to the polygon. This involves bounding the range of possible intersection points between two line segments with some uncertainty. There has been much work done in this area for example see [16] and the references contained therein.

We also noted in Section 4 that the performance of simplex is improved by switching to a new pivoting rule. However, we note that the pivoting rule we switched to is by no means state of the art. In fact, the steepest descent pivoting rule [12, 15] is the most often used pivoting rule in practice, and typically has an order of magnitude fewer pivots than that of Dantzig's rule. The rule itself takes some work to implement, but would increase the performance of COHO's simplex algorithm greatly. Alternatively, practical interior point methods such as Mehrotra's method also show better performance than a vanilla simplex implementation. There is currently no clear winner between Mehrotra's method and the steepest descent rule, as both have shown to perform well in practice on a variety of problems. Implementing these two methods would also be a great area for future work.

In Section 5, we noted that COHO was significantly IO bound by the Matlab frontend. In fact, the frontend accounts for up to roughly 90% of the end to end performance of COHO, which is unreasonably high. There are several ways to address this, the simplest of which is to simply write a proper communication pipeline between the Java backend and the Matlab frontend, either in Java or C++. Such a pipeline would also improve the performance of parallel COHO, as the current COHO architecture relies on global variables being set during the problem distribution phase, and cannot properly utilize the asynchronous capabilities of Matlab's built-in parallel processing functions. Such a change would time consuming, but not be hard to make, and is likely the most effective optimization for the current version of COHO.

In Section 6, we noted that a preliminary version of parallel COHO had been produced for linear models. Parallel COHO is still in its infancy and has much potential and future work ahead. Although linear systems was stated to be one area where COHO can be parallelized, there is much code within COHO that could be parallelized as well, such as the geometry operations where each projection fast is time advanced independently. Furthermore, parallel COHO has only ever been tested on one problem: the SpaceEx helicopter example. It would be interesting to see how COHO performs on other linear systems as well as on machines with a larger number of cores.

8 Conclusion

In this project, we performed an extensive profiling of the COHO reachability tool on several test problems, resulting in numerous optimizations in COHO's frontend and backend. Our optimizations improved the performance of COHO by up to a factor of 2.8 on the Java back end, up to a factor of 1.4 when using COHO through the Matlab interface.

Furthermore, we optimized and tested a parallel version of COHO for linear systems that shows a speed up factor of over 20 when tested on a high dimensional linear hybrid automata.

9 Acknowledgment

We thank Prof. Mark Greenstreet, for his support and generous extension on the due date of this project, and Chao Yan for helpful examples and discussions.

References

[1] VisualVM Java profiler. `https://visualvm.java.net/profiler.html`.

[2] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[3] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inf. Process. Lett.*, 9(5):216–219, 1979.

[4] Robert G. Bland. New finite pivoting rules for the simplex method. *Math. Oper. Res.*, 2(2):103–107, 1977.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[6] George B. Dantzig. *Linear programming and extensions*. Princeton Landmarks in Mathematics. Princeton University Press, Princeton, NJ, corrected edition, 1998.

[7] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 379–395, 2011.

[8] Philip E. Gill, Walter Murray, Michael A. Saunders, and Margaret H. Wright. A practical anti-cycling procedure for linearly constrained optimization. *Math. Programming*, 45(3, (Ser. B)):437–474, 1989.

[9] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[10] Mark R. Greenstreet and Ian Mitchell. Integrating projections. In *Hybrid Systems: Computation and Control, First International Workshop, HSCC'98, Berkeley, California, USA, April 13-15, 1998, Proceedings*, pages 159–174, 1998.

[11] Mark R. Greenstreet and Ian Mitchell. Reachability analysis using polygonal projections. In *Hybrid Systems: Computation and Control, Second International Workshop, HSCC'99, Berg en Dal, The Netherlands, March 29-31, 1999, Proceedings*, pages 103–116, 1999.

[12] Paula M. J. Harris. Pivot selection methods of the devex LP code. *Math. Program.*, 5(1):1–28, 1973.

[13] Marius Laza. A robust linear program solver for projectahedra. Master's thesis, The University of British Columbia, 2001.

[14] MATLAB. *Version 8.3.0 (R2014a).* Natick, Massachusetts, 2014.

[15] Jorge Nocedal and Stephen J. Wright. *Numerical optimization.* Springer Series in Operations Research and Financial Engineering. Springer, New York, second edition, 2006.

[16] Joab Winkler and Mahesan Niranjan, editors. *Uncertainty in geometric computations*, The Kluwer International Series in Engineering and Computer Science, 704. Kluwer Academic Publishers, Boston, MA, 2002. With 1 CD-ROM (Windows, Macintosh and UNIX).

[17] Chao Yan. Personal communication.

[18] Chao Yan. Coho: A verification tool for circuit verification by reachability analysis. Master's thesis, The University of British Columbia, August 2006.

[19] Chao Yan. *Reachability Analysis Based Circuit-Level Formal Verification.* PhD thesis, The University of British Columbia, 2011.

[20] Chao Yan. *CohoReach: Coho Reachability Analysis Tool*, 2015. `https://github.com/dreamable/cra/tree/fcra`.

[21] Chao Yan, Mark R Greenstreet, and Marius Laza. A robust linear program solver for reachability analysis. In *Proceedings of the First International Conference on Mathematical Aspects of Computer and Information Sciences (MACIS)*, pages pp231–242, Beijing, China, July 2006.