Autobahn Minimizer

Marilyn Sun
Tufts University
marilyn.sun@tufts.edu

Kathleen Fisher
Tufts University
kfisher@eecs.tufts.edu

Abstract

While lazy evaluation has many advantages, it can result in serious performance costs. Fortunately, Haskell allows users to enforce eager evaluation at certain program points by inserting bangs, which are strictness annotations. However, manual placement of bangs is both labor intensive and difficult to reason about. The Autobahn optimizer automatically infers bang patterns that improve runtime performance using genetic algorithms. However, Autobahn often generates very large numbers of bangs for each program. This is an issue for the user because each bang that cannot be deemed safe by the GHC compiler requires manual inspection to prevent the bang from introducing program non-termination.

This paper presents an improved version of Autobahn, which uses GHC profiling feedback to reduce the number of unnecessary bangs generated. AUTOBAHN 2.0 adds a pre-search phase before the genetic algorithm to adjust the search space size, and a post-search phase at the end to individually test and remove useless bangs. On average, the pre-search phase alone was able to eliminate 63 locations for potential bang placement per 100 LOC, and reduced the number of bangs eventually generated by 5 bangs per 100 LOC. Overall, AUTOBAHN 2.0 reduced the number of bangs generated from 11 bangs to 1 bangs per 100 LOC, while only slowing program runtime by 3%. Autobahn used in conjunction with the minimizer allows users to obtain faster versions of their programs without the burden of manually checking through large amounts of bangs.

ACM Reference format:
Marilyn Sun and Kathleen Fisher. 2018. Autobahn Minimizer. In *Proceedings of 11th ACM SIGPLAN International Haskell Symposium, St.Louis, MO, United States, September 27-28, 2018 (Haskell'18),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Haskell'18, September 27-28, 2018, St.Louis, MO, United States
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Lazy Evaluation

Lazy evaluation is a property of Haskell that improves program efficiency and provides programmers the ability to use infinite data structures. Under lazy evaluation, expressions are only evaluated when their values are needed. Every unevaluated expression is stored in a *thunk*, and its evaluation is delayed until another expression demands the value of the current one. Most of the time, this greatly improves program performance as it avoids wasting time on evaluating unnecessary expressions. It also means that users can operate on infinite data structures by only evaluating finite portions of it.

While lazy evaluation reaps many benefits, it can also create serious performance slow downs when too much memory is allocated to a large number of thunks. We can reduce thunks by avoiding creating them for expressions that we know will eventually be evaluated. To avoid the creation of a thunk, programmers can insert strictness annotations such as bang patterns at a certain program point to enforce eager evaluation. However, programmers need to distinguish program points that will benefit from eager evaluation from program points that do not need to be evaluated or will not terminate when evaluated. This task is difficult and often reserved for expert Haskell programmers.

1.2 Autobahn

AUTOBAHN is a Haskell optimizer that allows programmers to reduce thunks in their program by automatically inferring strictness annotations. Users provide AUTOBAHN with an unoptimized program, representative input, and an optional configuration file to obtain an optimized version of the program over the course of a couple of hours.

AUTOBAHN uses a genetic algorithm to randomly search for beneficial locations to place bangs in the program. The genetic algorithm iteratively measures the performances of a series of candidate bang placements. Candidates that improve upon the original program's performance are preserved, and candidates that trigger non-termination or worsen program performance are eliminated. AUTOBAHN eventually returns the user with a list of well performing candidates, ranked by how much they improve program performance. Users can then inspect the candidate bang placements, and decide if they want to apply one of them to the program.

1.3 Too Many Bangs

Candidates should be inspected before being applied because they can potentially introduce program non-termination when new input is given. Because Autobahn measures performance using representative input, the resulting candidate is optimized for that specific input. If a new type of input is supplied to the program after it is being optimized, it could result in different behavior at program points that cause the program to run forever. We can run GHC's static analysis after Autobahn to check for safe bangs, but it on average only marks 10% of bangs as safe. This is because the analysis is necessarily conservative to guarantee termination on all inputs.

In reality, bangs only need to be safe on inputs that the user will run it with. Because only the user will know the range of potential types of input, only the user can inspect candidate bang placements and decide if they are safe to be applied. However, users face a time consuming task of inspecting a large number of bangs when Autobahn generates too many bangs in a candidate. This occurs when the random genetic algorithm places many bangs throughout the program, including those that do not contribute much to program performance improvement. On average, Autobahn generates 24 bangs per 100 lines of code in its best performing candidates, and the user must manually inspect every one of those.

1.4 Autobahn 2.0

This paper presents an improved version of Autobahn that aims to reduce the number of generated bangs. We refer to the original Autobahn optimizer as Autobahn 1.0, and the improved version for bang minimization as Autobahn 2.0. A *pre-search* phase and *post-search* phase are each added before and after Autobahn 1.0 to locate and eliminate unnecessary bangs using GHC profiling. GHC profiles show the amount of runtime and memory each location in the program used. The pre-search phase adjusts the number of files that Autobahn 1.0 optimizes for within a single program. Autobahn 1.0 is instructed to optimize files that contain locations that cost a large amount of resources according to the GHC profile, and to avoid optimizing files that do not contain costly locations. After Autobahn 1.0 runs, the post-search phase individually tests each produced bang that falls within a costly location. Bangs that produce an insignificant impact on improving program performance are eliminated. On average, the addition of these two phases allows Autobahn 2.0 to produce 90.9% fewer bangs for a user to inspect than running Autobahn 1.0 alone would produce, while maintaining similar runtime performance. In this paper, I

- demonstrate the effectiveness of the pre-search phase on 20 programs from the NoFib benchmark suite that have had at least one file removed from consideration for optimization. On average the pre-search phase reduced 63 potential bang locations per 100 LOC, and resulted in mean bang reductions of 63.38%.
- show that Autobahn 2.0 applied to the NoFib benchmark suite on average reduced the number of bangs generated from 11 bangs per 100 LOC to 1 bang per 100 LOC, while only increasing the optimized runtime by 3%.
- use Autobahn 2.0 in a case study to optimize the performance of the `gcSimulator` garbage collector simulator. While the program runtime slowed down by 15%, the number of bangs generated decreased by 81.9%.
- apply Autobahn 2.0 in a second case study to show that it can preserve the application-specific annotations inferred by Autobahn 1.0 for the Aeson library.

2 Background

2.1 GHC Profiling and Cost Centres

To allow users to better understand where their program spends the most time on, GHC provides a time and memory profiling system. The system adds annotations to the user's program and generates a report detailing the amount of time, memory allocations or heap usage each location used.

To generate these profiles, users simply run their program after re-compiling it with the profiling option and choose either a time and allocation or heap profile to generate, as well as the method in which the profiling system adds annotations. While users have the option to manually specify annotations, Autobahn 2.0 uses the `-prof -fprof-auto` option, which automatically adds an annotation around every binding that is not marked INLINE in the program.

In the profile, these annotations are represented as cost centres with a certain cost associated with each of them. These costs indicate how much time or memory resources each cost centre used as a percentage of the whole program's resources.

In order to minimize the number of bangs in a program while maintaining similar program performance, we need to preserve the bangs in the most costly cost centres and eliminate those located in the less costly cost centres. The minimizer identifies a cost centre that consumes more than a *hotSpotCost*threshold percentage of the overall program runtime as a *hot spot*. A cost centre that does not consume more than the *hotSpotCost* threshold is a *cold spot*. Currently, we set the *hotSpotCost* threshold to 6%, although that threshold can be adjusted. As the threshold increases, fewer bangs are preserved at the risk of a higher possibility of compromised program performance.

2.2 Genes and Chromosomes

Cost centre profiling provides guidance for the otherwise random search that Autobahn performs using genetic algorithms. In the algorithm, any program source location where a bang may be added becomes a *gene* that can be turned on or off. A chromosome is composed of all of the genes within a program and represented as a fixed-length bit vector, in which the bit value indicates the presence or absence of a bang. Since a program is a collection of source files, it is represented as a collection of bit vectors, or *chromosomes*.

2.3 Autobahn's Genetic Algorithm

Autobahn's genetic algorithm evaluates and manipulates randomly generated chromosomes. It repeatedly generates new chromosomes before measuring their performance using a fitness function. We call a chromosome that either significantly slows down program performance or causes non termination an *unfit* chromosome. If the fitness function determines that a chromosome is unfit, the chromosome is immediately discarded. If the fitness function determines that a chromosome behaved well, the chromosome is deemed *fit* and kept for future rounds of generation.

For each round of chromosome generation, Autobahn introduces randomness by performing either a mutation or a crossover. A mutation flips a gene in the chromosome whenever a randomly chosen floating point number exceeds the *mutateProb* threshold. A crossover combines two chromosomes by randomly picking half of the genes from each parent chromosome. For either of these random operations, Autobahn uses a unique number generator each time to guarantee randomness.

2.4 Representative Input

To run Autobahn 1.0, users need to provide representative input to their program. The input should be short enough for Autobahn to finish execution in a reasonable amount of time while still be long enough for Autobahn to measure noticeable time improvements if there are any. Ideally, representative input should also be as similar to the typical use case of the program as possible to reduce chances of unexpected behavior after optimization when using different types of program input.

Similarly, the quality of representative input impacts the quality of Autobahn 2.0's performance as well. Different types of input may generate wildly different results in GHC profiles, which are heavily relied on by the minimizer later on. Therefore, the user must carefully choose their program's representative input.

3 Autobahn 2.0

3.1 Why Too Many Bangs Are Generated

The first step to eliminating bangs is to identify categories of bangs and hypothesize the reason why each category exists. A *dangerous* bang is a bang that can significantly slow down program runtime or cause program non-termination. A *useful* bang improves program performance, and a *useless* bang neither improves nor worsens program performance.

While an unfit chromosome may perform poorly as a whole, it can contain a mixture of dangerous, useful and useless bangs. Autobahn 1.0 handles unfit chromosomes by discarding them as a whole, but fails to provide a method of isolating the specific dangerous bangs in an unfit chromosome. It is necessary to isolate and remove dangerous bangs because they might otherwise reappear in later generations as a result of random mutation.

Well performing, or *fit* chromosomes, also face a similar issue. Autobahn 1.0 handles *fit* chromosomes by preserving the entire chromosome, without separately identifying the useful bangs from the useless ones. This is problematic for two reasons. Firstly, we might lose useful bangs in future rounds of generations because we cannot track them and guarantee that random methods of mutation and crossover will preserve them. Secondly, useless bangs could survive by being grouped with useful bangs even though they should be eliminated. The accumulation of such bangs can dramatically increase the amount of bangs in a program, leaving it up to the user to identify potentially unsafe bangs from the safe ones.

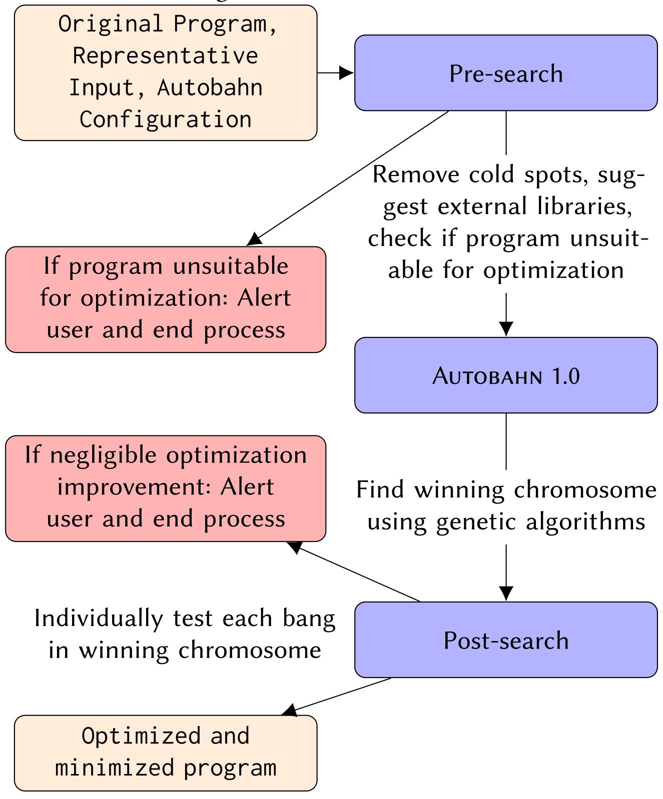
We hypothesize that Autobahn 1.0 generates copious amounts of bangs because it is incapable of identifying categories of bangs within the same chromosome. The further addition of randomness means that the entire chromosome is repeatedly searched as the search space is never definitively reduced. Because there is a fixed number of genes in a program, the search space for the genetic algorithm is also equivalently fixed. Therefore, as the program source code increases in size, the algorithm also generates significantly more bangs as chromosomes increase in size.

3.2 The Solution

Autobahn 2.0 uses GHC profiling to do what Autobahn cannot: isolate portions of a chromosome by their individual contributions to program performance. Cost centres not only break down a chromosome into smaller portions by source code bindings, but their associated costs also imply how likely a bang placement will affect program performance. If executing code at a hot spot occupied a significant portion of the overall program runtime, then a bang-induced change in performance at the hot spot will likely significantly affect overall runtime.

There are two ways to apply GHC profiling information to reduce the number of bangs generated by Autobahn. The first way is by reducing the previously fixed size of the initial chromosome. Because Autobahn's search space is directly correlated to the size of the program source code, we can reduce the search space by eliminating all genes in cold spots prior to Autobahn's optimization. Since useless bangs are most likely later generated and located within cold spots, early elimination of genes in cold spots is beneficial.

However, hot spots can also contain a mixture of useful and useless bangs as well. To definitively eliminate useless bangs and preserve useful bangs in hot spots, we can individually isolate and measure the performance of each bang in each hot spot. The effects of combining bangs is hard to predict, but the permutation size of all possible types of combinations of the remaining bangs can grow very large. To simplify the process, we adopt the method of individually turning off one bang at a time to test. We can exhaustively test each bang because the size of the set of hot spots in a program is limited. It is then necessary to include Autobahn 1.0's results, because the genetic algorithm would've randomly found a winning combination of bangs as a starting point for testing. Both of these methods are later explained in full detail, and a diagram of the workflow is shown below.


Original Program, Representative Input, Autobahn Configuration
Pre-search
Remove cold spots, suggest external libraries, check if program unsuitable for optimization
If program unsuitable for optimization: Alert user and end process
Autobahn 1.0
Find winning chromosome using genetic algorithms
If negligible optimization improvement: Alert user and end process
Individually test each bang in winning chromosome
Post-search
Optimized and minimized program

3.3 Autobahn Coverage

By default, Autobahn optimizes all files in the program directory. Users can specify optimization coverage by manually adding or removing file paths while configuring Autobahn. Although Autobahn does not consider external libraries imported in the source code, users can manually add local copies of external libraries in the program directory for optimization.

However, just as manually reasoning about the placement of bangs is difficult, users generally find it difficult to reason about which files should or should not be optimized. By examining GHC profiles, we can have a much better idea of which files to eliminate based on whether or not they contain hot spots. Combining this knowledge with the configuration option of configuring Autobahn coverage, we can now manipulate the initial chromosome size by units of file sizes.

3.4 Pre-search Profiling

Pre-search profiling does more than just reducing existing search space. Instead, it guides, redirects and expands the current search space to maximize efficiency and performance prior to Autobahn 1.0's optimization. Because chromosome sizes directly influence the number of generated bangs, search space manipulation can minimize the possibility of generating uselessor dangerousbangs and maximize the chances of generating usefulones.

To reduce the initial search space, the minimizer begins by generating a GHC time and allocation profile for the unoptimized program with user provided representative input. Then, the files that contain at least one hot spot are identified. Files that do not contain hot spots are eliminated from the Autobahn coverage of files. Autobahn then optimizes the program as usual, except using a much smaller set of files and thus chromosomes to begin its genetic algorithmic search.

There are three important impacts that pre-search profiling creates. First of all, it greatly reduces the number of bangs Autobahn generates by reducing the initial search space. Secondly, it is capable of identifying programs that are potentially unsuitable for optimization using Autobahn. If a program contains a large set of cost centres that all contribute minimally to program runtime, there may not exist a single location in which placing a bang will make a significant difference in program runtime. If the minimizer identifies a program that only contains cold spots, it will alert the user and save them the time and effort of running Autobahn when they will most likely see minimal performance improvements. Lastly, if a hot spot is located in an external library file, the minimizer can suggest users which external libraries to add to the Autobahn coverage for better optimization results.

3.5 Post-search Bang Elimination

After Autobahn 1.0 optimizes the search space and determines a winning chromosome, Autobahn 2.0 once again uses GHC profiling information to reduce the number of bangs in the winning chromosome. It begins by mapping each gene in the winning bit vector to their corresponding cost centres.

For each gene in the bit vector, the post-search phase filters out all of those that are already turned off and keep them turned off. It then examines genes that are turned on but do not fall within a hot spot, and turns them off before filtering them out as well. We can turn them off because those genes are most likely useless bangs.

The remaining genes are the interesting ones that both contain a bang and fall within a hot spot. These genes require further testing because even though hot spots are likely to significantly affect overall performance when their own runtimes are reduced, we are unsure if the cost centre runtime was reduced in the first place. That is, placing a bang in a cost centre may not always improve the actual cost at the cost centre. Therefore, genes within hot spots that cannot be improved through the use of bangs are also useless and should be discarded.

3.6 Testing hot spots

A convenient fact about testing these remaining genes that are both turned on and within a hot spotis that they are usually so few in number that it is possible to exhaustively search them. The minimizer tests them by isolating and turning off each gene while keeping all other remaining genes on. It then measures program runtime and compares it to the program performance of the winning chromosome determined by Autobahn.

If the absence of this gene slows down the program by an *absenceImpact* threshold, this gene is determined to be useful and is kept in the pool of remaining genes. If the gene's absence does not slow down the program by at least the *absenceImpact* threshold, the gene is deemed useless and discarded. The *absenceImpact* threshold is adjustable and currently set to 5%. The post-search phase repeats this process for every gene to be tested, and the minimization result is the combination of bangs in the pool of remaining genes by the end of testing.

4 Implementation

4.1 Program Architecture

The addition of search space reduction and post-search bang reduction alters the original program architecture of Autobahn 1.0. Prior to optimization, the original program is first profiled and evaluated for search space manipulation in the pre-search phase. pre-search builds the user's program with profiling enabled, and runs the unoptimized version with the user provided representative input to obtain a time and allocation profile. The profile is only generated once, and the rest of of Autobahn 2.0 refers to the same profile throughout the entire program. Depending on the location of hot spots indicated by the profile, the program's source files will either be automatically reduced or manually expanded by the user.

Then Autobahn 1.0 is executed using the same genetic algorithm to find a winning chromosome. It uses the *haskell-src-exts* parser to parse source files and identify genes, then applies a genetic algorithm with a fitness function to search for the best performing chromosome.

The resulting chromosome is further tested and reduced using GHC profiling information in the post-search bang reduction phase. After an initial pass of elimination to get rid of all turned off genes and genes located in cold spots, we individually test the impact of the absence of a turned on gene in each hot spot. If a gene meets the *absenceImpact* threshold, it is kept in future rounds of testing and will remain in the final combination of bangs for the optimized program. If a gene does not meet the *absenceImpact* threshold, it is removed for future rounds of testing and will not appear in the final combination.

Bangs in hot spots are tested in order of decreasing costs. While we recognize that the order in which we test them may affect their performance, it is simply too time consuming to test every possible combination of bangs in every possible order. (insert permutation calculations of max possible no. of combinations based on 6% *hotSpotCost* here) Therefore, we chose to only consistently test each individual bang once in order of decreasing costs for simplicity.

Finally, the post-search phase returns the final combination of bangs that have survived each round of testing. If Autobahn 1.0 failed to find a chromosome that improved program runtime by 6% to begin with, then the post-search phase will refuse to minimize because the insignificant performance improvements indicates that users are better off keeping the original unoptimized program instead.

4.2 Running Autobahn 2.0

A user runs Autobahn 2.0 the same way as they would run the Autobahn 1.0. The user provides a copy of their program source code, representative input, and an optional Autobahn configuration file. Because both search space reduction and minimization after Autobahn typically do not require a significant amount of time, the user should barely notice an increase in the amount of time needed for optimization.

If Autobahn 2.0 successfully ran, the user can find the minimized source code in the same project directory along with the usual Autobahn survivor and results directories. If pre-search profiling detected that the program is unsuitable for optimization, or if Autobahn 1.0 failed to significantly optimize the program, then Autobahn 2.0 would warn the user and halt the optimization process. If external libraries could be added to Autobahn's coverage to potentially boost optimization performance, the pre-search phase would alert the user and continue to optimize.

4.3 Source Locations

Because each bit in a bit vector represents a gene in a chromosome, we modified the bit vector to indicate which cost centre each bit was located in. Cost centres are uniquely identified by source location in source files. We mapped each bit in Autobahn's bit vectors to its corresponding source line. To turn the bangs in a hot spot on or off, we can traverse the bit-location vector and manipulate the bits that are tagged with source lines that fall within the range of that hot spot's source location.

4.4 Removing Illegal Genes

Unfortunately, the *haskell-src-exts* parser that Autobahn 1.0 uses incorrectly identifies the left hand side of bindings within instance declarations as potential locations to place bangs. For that reason, files that contain instance declarations have been previously avoided and left unoptimized when testing Autobahn 1.0. We wanted to support the optimization of these files in Autobahn 2.0, so we removed any randomly generated illegal bangs prior to each round of fitness evaluation in the genetic algorithm. The rest of the genetic search and Autobahn 2.0 runs identically as before.

To keep track of whether a bang is legal, we used a validity-indicating boolean vector to represent whether each gene in source code is legal. Prior to inserting bangs into a program, Autobahn 2.0 would check the validity of a gene against the boolean vector to make sure that the bang is located in a legal location.

Generically traversing the parser generated AST using boilerplate code fails to identify illegal genes, so we needed to manually traverse it to construct the validity vector. As we traversed the tree, we kept track of whether a left hand side binding is within an instance declaration. If so, then that binding is an illegal bang location and is marked as a false boolean value. All other legal bang locations are marked as a true value.

Autobahn 2.0 successfully uses this method to avoid inserting bangs into illegal locations after being misguided by the parser. As a result, it allows us to optimize files that include instance declarations, which we previously avoided altogether when using Autobahn 1.0.

5 Evaluation

5.1 Experiment Setup

All versions of Autobahn were compiled with GHC version `8.0.2`. The NoFib benchmarks were also compiled with GHC version `8.0.2`, `-XBangPatterns` and enabled profiling along with NoFib's default flags. Our research did not test the certain benchmarks in the NoFib suite that failed to compile on their own.

5.2 Pre-search Search Space Reduction

To test how much search space can be reduced by the pre-search phase of Autobahn 2.0, we ran Autobahn 1.0 with pre-search profiling on the NoFib benchmark suite. To account for fluctuation, we took the mean of running the program ten times on the benchmark suite. All runs were optimized on runtime only, and the *hotSpotCost* and *absenceImpact* thresholds were both set to 6%.

Figure 1 includes results from the 20 benchmarks that had at least one file that was eliminated during the pre-search phase. The number of eliminated genes show the number of potential bang locations that was eliminated before Autobahn 1.0 ran. Because most benchmarks do not have bangs in the original versions of their programs, the number of original bangs in Figure 1 is always 0.

The `anna`, `expert`, and `symalg` benchmarks are particularly interesting because Autobahn 1.0 consistently failed to find winning chromosomes for them, so 0 Autobahn bangs were generated. However, after refining the search space using pre-search, Autobahn 1.0 was able to better search the space and find meaningful bangs as indicated by the number of Pre-Autobahn bangs generated.

Figure 2 shows the corresponding runtime performance of results generated by Autobahn 1.0 and Autobahn 1.0 with the pre-search phase. The graph shows that even when a large number of genes are eliminated prior to optimization, the optimizer is still able to find useful bangs that result in similar runtime improvement.

5.3 Pre-search File Elimination

We have found six benchmarks in the NoFib suite that the pre-search phase identifies as unsuitable for optimization when the *hotSpotCost* threshold is set to 6%. These are `awards`, `callback001`, `callback002`, `mutstore2`, `sorting`, and `threads007`. As expected, when attempting to optimize `awards` and `threads007`, Autobahn consistently fails to optimize and returns 1.0, indicating that it cannot do better than the original program. However, Autobahn was able to successfully optimize the other programs. Through inspection, we concluded that `callback002` would've benefited from a lower *hotSpotCost* threshold as its most costly `hot spot` takes up 3.9% of the program runtime. Both `callback001` and `threads007` would've benefited from inspecting heap profiles instead of time and allocation profiles as the costs associated with their hot spots were noticeably larger in heap allocations while remaining insignificant in runtime costs. `mutstore2` is a program that's performance fluctuated wildly even without bangs in it. For example, its measured runtime was as low as 60% - 80% of its original runtime in one third of the experiments we ran with no bangs in the program. Therefore, the optimization results were likely skewed by the fluctuating runtime.

5.4 Pre-search File Addition

We also ran experiments using the Aeson parser library that handles JSON files using either the `validate` or `convert` driver programs. `validate` simply checks if the file is written in valid JSON syntax, and `convert` actually converts file input into a Haskell data structure.

While running the pre-search phase on the Aeson library, we were suggested by the program to include files from the external `Data/Aeson` library in the Autobahn optimization coverage because the `Data/Aeson/InternalTypes.hs` file

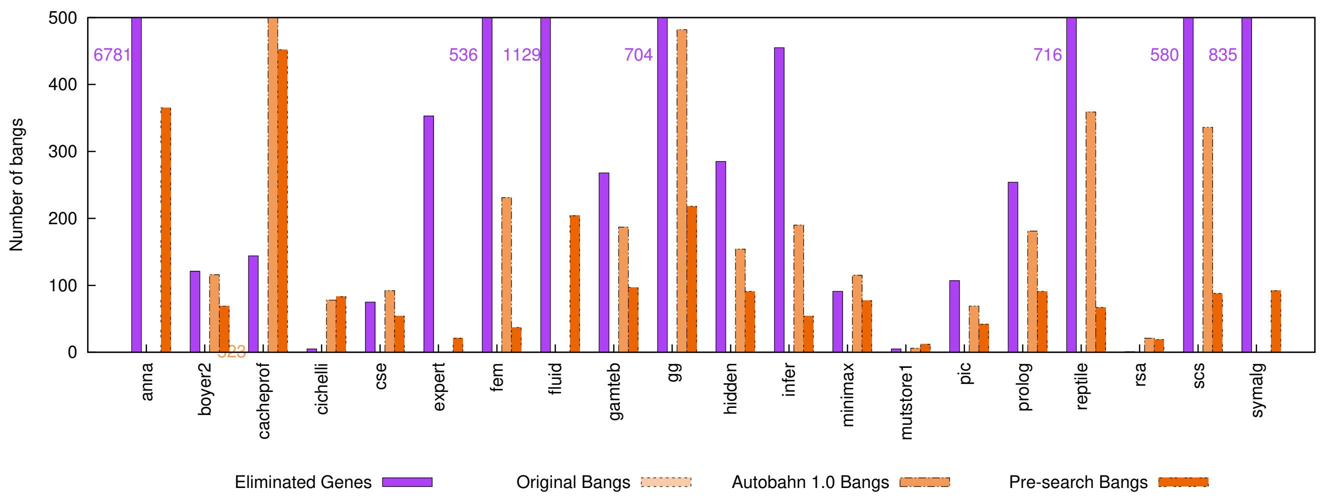


Figure 1. Number of bangs generated by AUTOBAHN 1.0 vs. pre-search phase combined with AUTOBAHN 1.0 compared across 20 benchmarks. Columns that exceed the maximum axis value are labelled with their actual values.

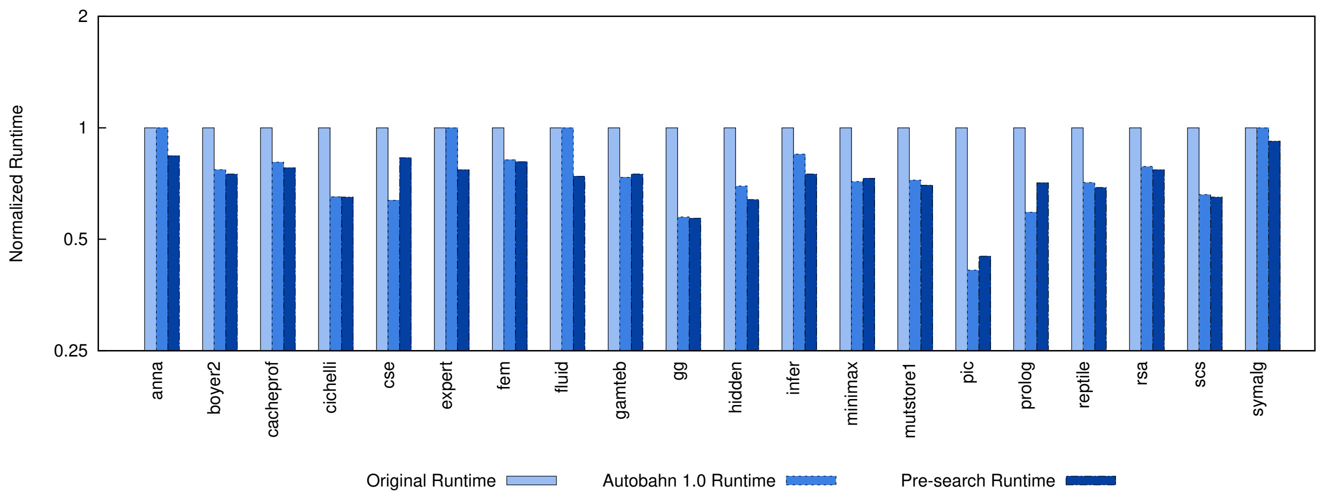


Figure 2. Normalized runtime of AUTOBAHN 1.0 results vs. pre-search phase combined with AUTOBAHN 1.0 results across 20 benchmarks. Columns that exceed the maximum axis value are labelled with their actual values.

included multiple hot spots. However, `InternalTypes.hs` already included manually inserted bangs by its author. Therefore we ran experiments using a version of `InternalTypes.hs` with no bangs in it to see if we could recreate the manually inserted bangs. If so, library authors could run Autobahn to place bangs in their files instead of manually doing so.

Version	Driver	Normalized Runtime
Original	value	1
	json	1
Pre-optimization	value	0.73
	json	0.66
AUTOBAHN 1.0	value	0.56
	json	0.72

5.5 Post-search Bang Reduction

To test the efficiency of reducing bangs using the post-search phase, we compared the results of running only AUTOBAHN 1.0 with AUTOBAHN 1.0 and the post-search phase. Similarly, we took the mean of running the program ten times on the benchmark suite while optimizing on runtime only, and set both *hotSpotCost* and *absenceImpact* thresholds to 6%.

If AUTOBAHN 1.0 improves a benchmark's performance by at least 6% after optimization, we define the benchmark a successfully optimized. Figure 3 and Figure 4 include results from the 22 benchmarks that were successfully optimized.

Figure 3 shows that the number of bangs eliminated by the post-search is quite significant. Figure 4 shows the corresponding runtime performance of each program. In most benchmarks, the post-search phase does a little worse than running only Autobahn 1.0, because the *absenceImpact* threshold limits the remaining bangs to those that affect program runtime by at least 6%. If a user wants to maintain more similar runtime results, they can lower the *absenceImpact* threshold so the minimizer becomes less aggressive in bang elimination. That way, more bangs will be preserved, but runtime performance will improve.

The interesting results for programs *anna* and *fluid* show that while Autobahn 1.0 found bangs that triggered a 2.0 error code of program non termination, post-search bang elimination was able to get rid of the dangerous bangs that caused non termination.

5.6 Combining pre-search and post-search

5.7 Autobahn 2.0 On Larger Programs

We ran Autobahn 2.0 on the gcSimulator garbage collector to see how well it performs on larger programs. To keep optimization runtime within reasonable ranges, we used the first 1M of the batik trace file as the representative input. For gcSimulator, we lowered the *absenceImpact* threshold to 1% because it does not have many hot spots to begin with. Once Autobahn 2.0 was done optimizing, we tested the resulting program on larger trace file sizes of 100M and 500M.

The original Autobahn 1.0 was able to produce results that not only ran faster on representative input, but also on larger trace files as well. Autobahn 2.0 was also able to generate similar results with much fewer bangs.

Version	File Size (M)	Runtime	No.Bangs
Original	1	0.40	0
	100	43.13	0
	500	216.71	0
Autobahn 1.0	1	0.18	690
	100	14.19	690
	100	68.98	690
Autobahn 2.0	1	0.23	125
	100	15.75	125
	500	81.66	125

gcSim (both optimized and original) runs for waaaay too long on full batik trace, not sure how Remy did it. Here I'm doing increments of 1, 100, 500M instead. The full batik trace is 6184M

5.8 Autobahn 2.0 On Aeson

Claim: Minimizing Aeson can preserve different annotations in bang reduction in convert and validate.

I have the same results from above pasted here. A single experiment on Aeson takes up to 3 days to run because cabal compiling is so slow. Need to run more experiments for Aeson.

Version	Driver	Normalized Runtime
Original	value	1
	json	1
Pre-optimization	value	0.73
	json	0.66
Autobahn 1.0	value	0.56
	json	0.72

6 Related Work and Future Work

6.1 Autobahn 1.0 and Other Methods of Removing Laziness

The current strictness analyzer in GHC uses backward abstract interpretation to identify locations that can be eagerly evaluated. The analysis is approximate because the analysis is static. The analysis also conservatively binds locations as strict only if it can guarantee program termination because it is a part of the compiler. Autobahn has the advantage of both being dynamic and not needing to guarantee termination on all inputs as it is not a a part of the compiler. Instead, it allows users to decide the safety of suggested strictness annotations based on the intended application of the program. Other approaches to reduce laziness include Strict Haskell, which allows users to make entire modules strict rather than lazy by default using the -XStrict and -XStrictData language pragmas. Chang and Felleisen starts with a program written in a strict language, and inserts laziness annotations into it using dynamic profiling. It would be interesting to see if Chang and Felleisen's method could be applied to introduce laziness to Strict Haskell programs.

6.2 Profiling Haskell Programs

Apart from the profiling system that GHC provides, there exists a variety of other cost-based profiling systems for Haskell.

6.3 Autobahn 2.0 Improvements

For future developments, it would be worth exploring the additional use of heap profiles to locate hot spots instead of solely using time and allocation profiles. Furthermore, our experiments show that the ideal values for *hotSpotCost* and *absenceImpact* thresholds vary by program to program. Adopting more flexible thresholds that automatically adjust themselves based on the results of cost centre profiling might yield better results than using set values or asking users to provide them.

7 Conclusion

Laziness is a double edged sword: While it provides many benefits, excessive laziness often causes poor performance. Strictness annotations allow programmers to force eager evaluation, but its use is limited to programmers with extensive experience and high levels of expertise. Autobahn uses a genetic algorithm to automatically infer annotations

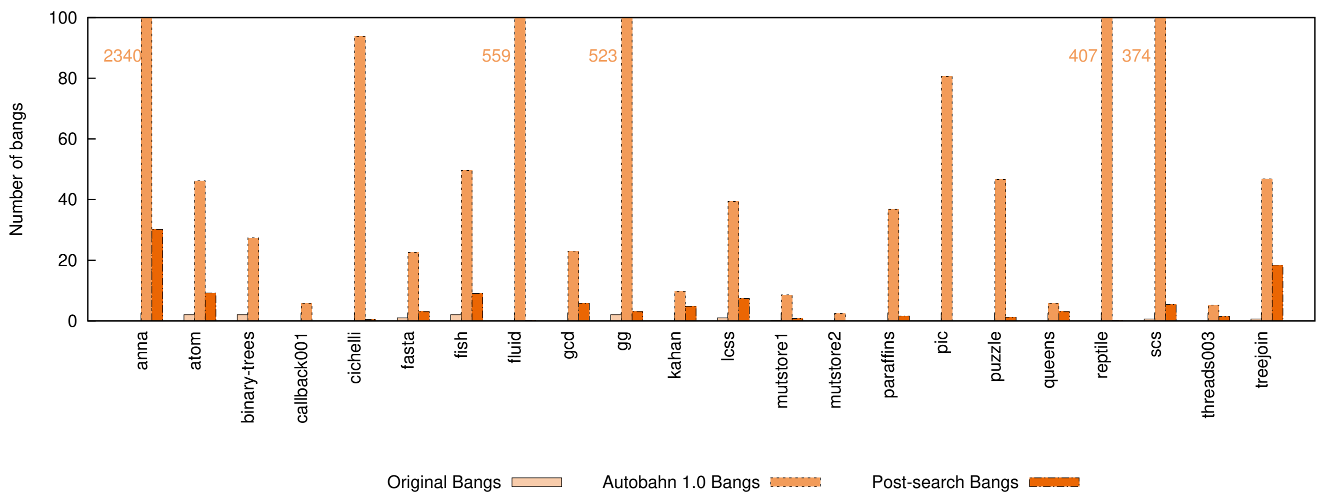


Figure 3. Number of bangs generated by AUTOBAHN 1.0 vs. post-search phase combined with AUTOBAHN 1.0 across 22 benchmarks. Columns that exceed the maximum axis value are labelled with their actual values.

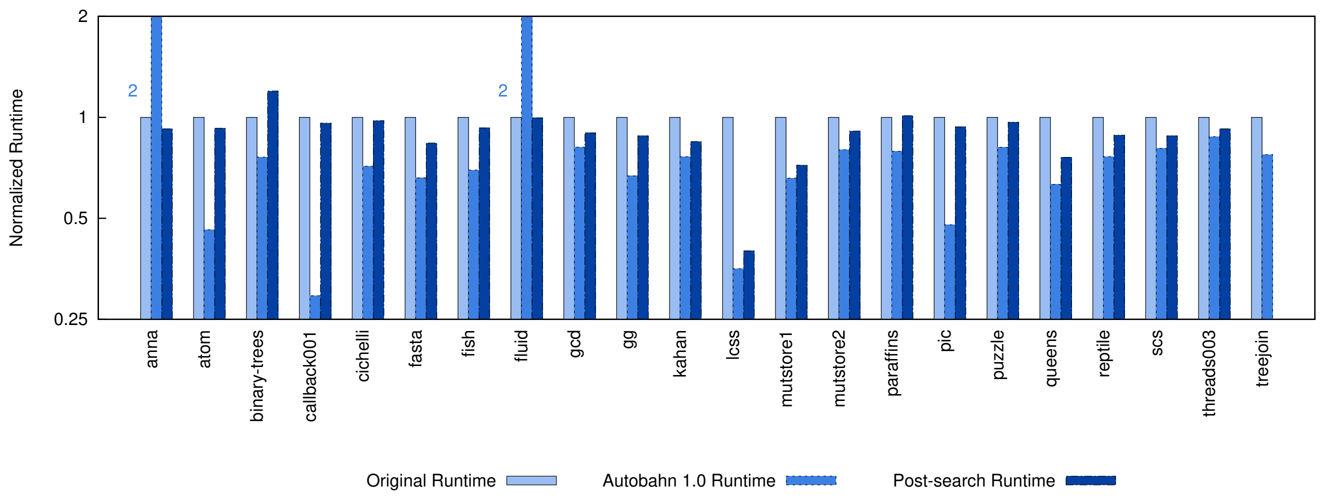


Figure 4. Normalized runtime of AUTOBAHN 1.0 results vs. post-search phase combined with AUTOBAHN 1.0 results across 22 benchmarks. Columns that exceed the maximum axis value are labelled with their actual values. Benchmarks that did not terminate are indicated with a 2.0 error code runtime.

for better program performance, but it often suggests too many bangs for users to inspect. We have built Autobahn 2.0, which uses GHC profiling feedback to perform search space manipulation to improve the efficiency of genetic algorithms, and eliminates bangs based on their associated performance costs. Experiments show that AUTOBAHN 2.0 - NoFib runtime improvement - NoFib Bang reduction - Results

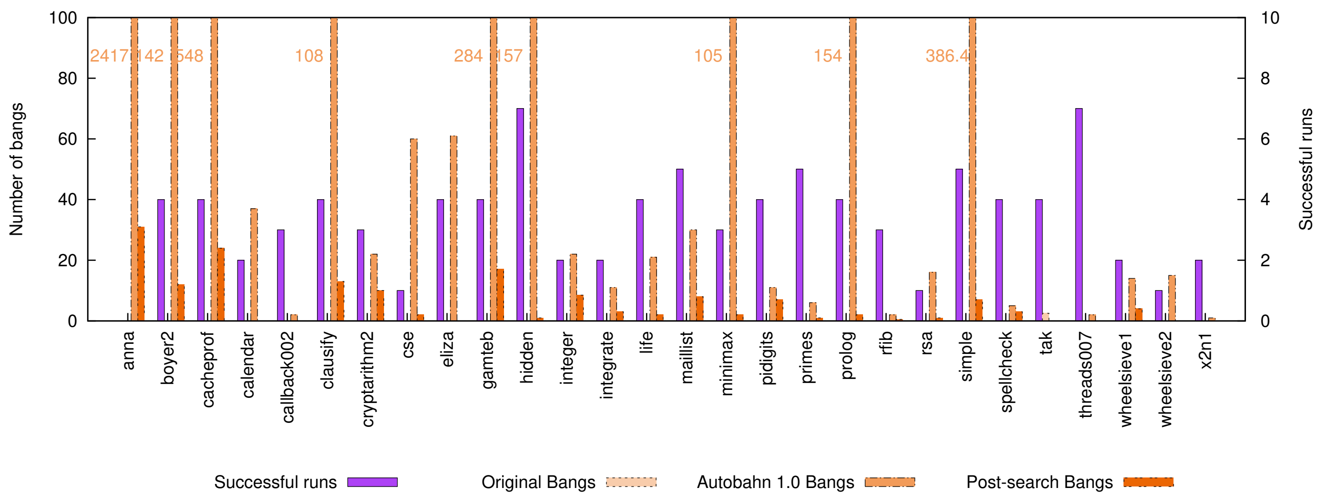


Figure 5. Number of bangs generated by Autobahn 1.0 vs. post-search phase combined with Autobahn 1.0 across 28 benchmarks. Success rate out of 10 runs is shown. Columns that exceed the maximum axis value are labelled with their actual values.

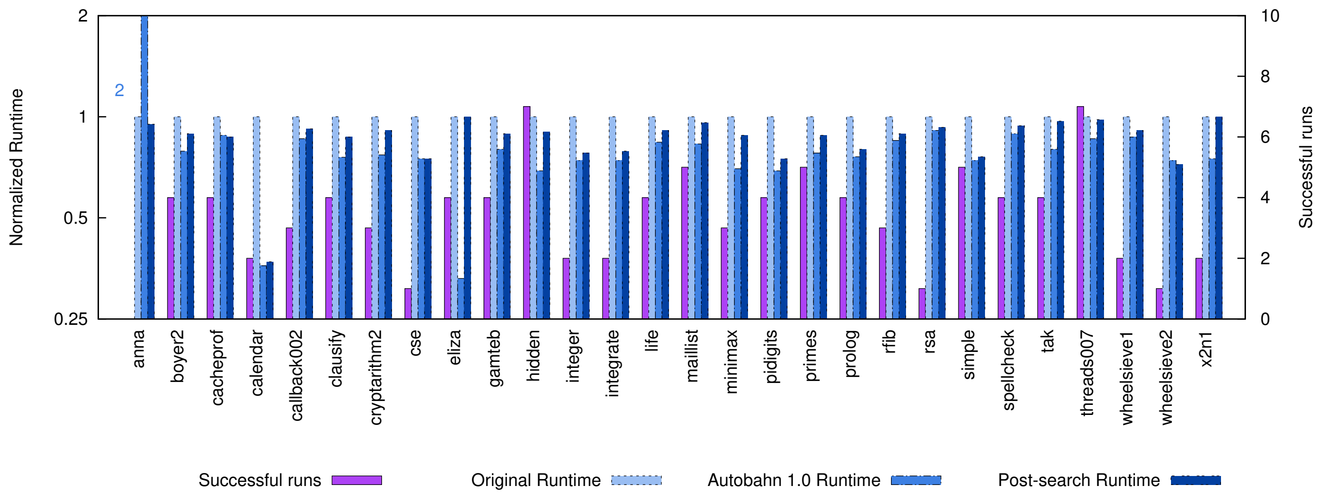


Figure 6. Normalized runtime of Autobahn 1.0 results vs. post-search phase combined with Autobahn 1.0 results across 28 benchmarks. Success rate out of 10 runs is shown. Columns that exceed the maximum axis value are labelled with their actual values. Benchmarks that did not terminate are indicated with a 2.0 error code runtime.

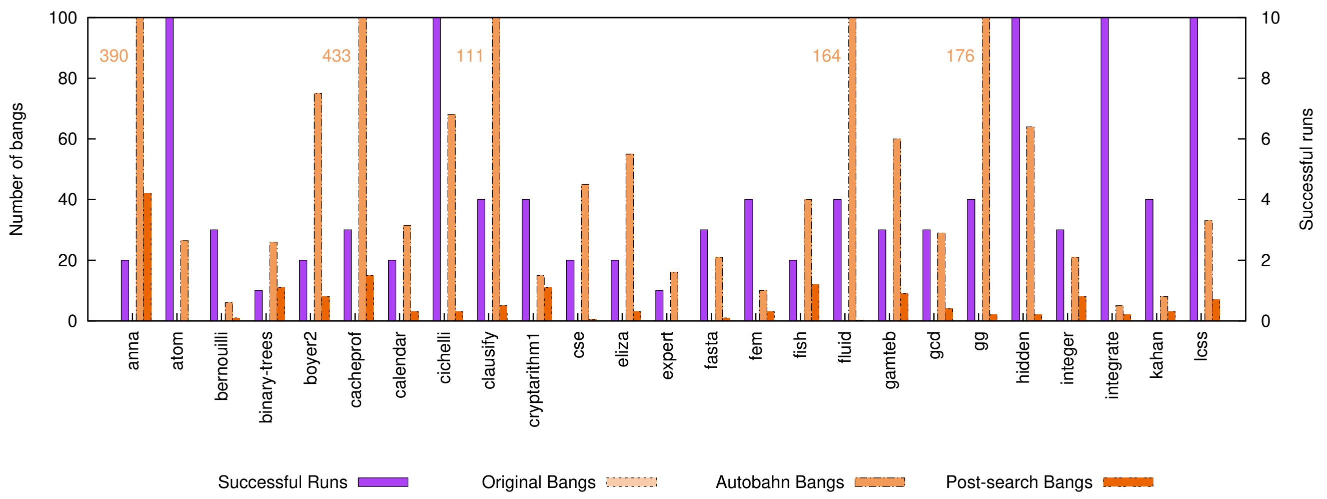


Figure 7. Number of bangs generated by Autobahn 1.0 vs. Autobahn 2.0 across 25 benchmarks. Success rate out of 10 runs is shown. Columns that exceed the maximum axis value are labelled with their actual values.

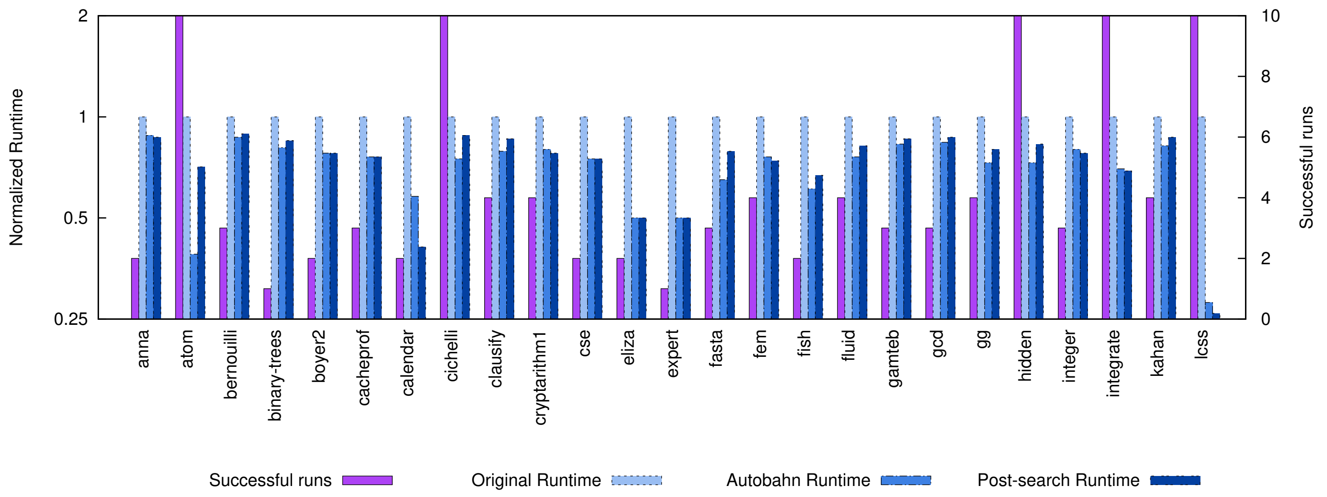


Figure 8. Normalized runtime of Autobahn 1.0 results vs. Autobahn 2.0 results across 25 benchmarks. Success rate out of 10 runs is shown. Columns that exceed the maximum axis value are labelled with their actual values. Benchmarks that did not terminate are indicated with a 2.0 error code runtime.

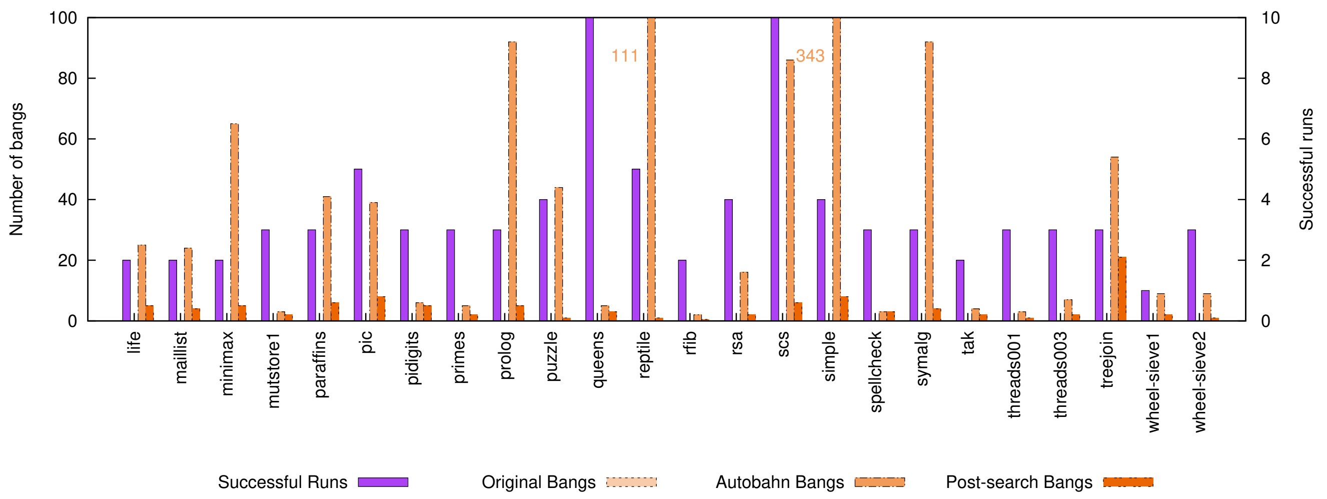


Figure 9. Number of bangs generated by Autobahn 1.0 vs. Autobahn 2.0 across 24 benchmarks. Success rate out of 10 runs is shown. Columns that exceed the maximum axis value are labelled with their actual values.

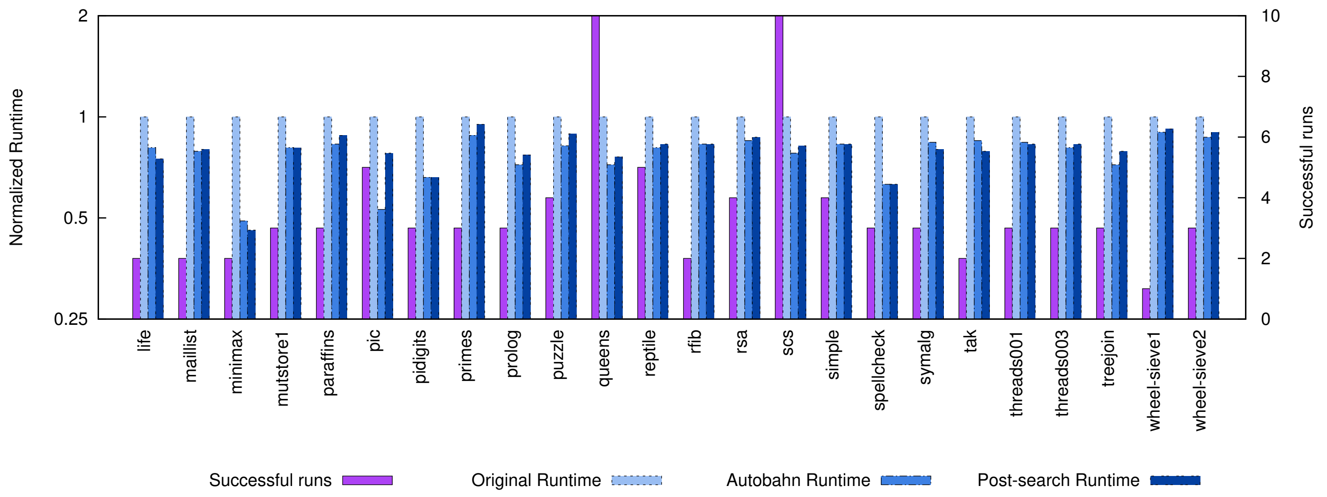


Figure 10. Normalized runtime of Autobahn 1.0 results vs. Autobahn 2.0 results across 24 benchmarks. Success rate out of 10 runs is shown. Columns that exceed the maximum axis value are labelled with their actual values. Benchmarks that did not terminate are indicated with a 2.0 error code runtime.