TUFTS UNIVERSITY

AUTOBAHN 2.0: Minimizing Bangs while Maintaining Performance

Author:
Marilyn SUN

Supervisor:
Kathleen FISHER

March 10, 2019

"Insert quote here."

Name

TUFTS UNIVERSITY

Abstract

Faculty Name
Tufts Department of Computer Science

Bachelor of Science

AUTOBAHN 2.0:
Minimizing Bangs while Maintaining Performance

by Marilyn SUN

While lazy evaluation has many advantages, it can result in serious performance costs. To alleviate this problem, Haskell allows users to force eager evaluation at certain program points by inserting strictness annotations, known and written as bangs (!). Unfortunately, manual bang placement is labor intensive and difficult to reason about. The AUTOBAHN 1.0 optimizer uses a genetic algorithm to automatically infer bang annotations that improve runtime performance. However, AUTOBAHN 1.0 often generates large numbers of superfluous bangs, which is problematic because users must inspect each such bang to determine whether it introduces nontermination or other semantic differences. This paper introduces AUTOBAHN 2.0, which uses GHC profiling information to reduce the number of superfluous bangs. Specifically, AUTOBAHN 2.0 adds a *pre-search phase* before AUTOBAHN 1.0's genetic algorithm to focus the search space and a *post-search phase* to individually test and remove bangs that have minimal impact. When evaluated on the NoFib benchmark suite, AUTOBAHN 2.0 reduced the number of inferred bangs by 90.2% on average, while only degrading program performance by 15.7% compared with the performance produced by AUTOBAHN 1.0. In a case study on a garbage collection simulator, AUTOBAHN 2.0 eliminated 81.8% of the recommended bangs, with the same 15.7% optimization degradation when compared with AUTOBAHN 1.0.

Acknowledgements

The acknowledgments and the people to thank go here, don't forget to include your project advisor...

Contents

List of Figures

List of Tables

For/Dedicated to/To my…

Chapter 1

Introduction

1.1 Too Many Strictness Annotations In Optimized Programs

Meet Pat and Chris. They recently implemented a Haskell program that they were incredibly excited about before they realized that the program ran too slowly. Pat and Chris' performance bug was a result of *thunk leaks*, in which Haskell stores unevaluated expressions as a part of its *lazy evaluation strategy*. To counteract thunk leaks, Haskell allows users to insert *strictness annotations* to enforce strict evaluation at various program points. After some research, Pat and Chris decided to optimize their program using AUTOBAHN 1.0, a Haskell optimizer that uses genetic algorithms to automatically infer strictness annotations.

Pat and Chris provided AUTOBAHN 1.0 with representative input to optimize their program with and allowed the optimizer to finish running overnight. In the morning, they were delighted to find that their program indeed ran much faster on the provided input than it originally did. But Pat and Chris wanted their program to run faster not only on representative input, but on all types of input that it might accept. Most importantly, Pat and Chris needed to ensure that their program at the very least ran safely, without non-termination or other incorrect behaviors, on all types of input.

Since only Pat and Chris know what other types of input their program might accept, they are tasked with manually inspecting every strictness annotation that AUTOBAHN 1.0 produced to make sure that all annotations will produce safe behavior on all inputs. But they soon realize that AUTOBAHN 1.0 produced hundreds of annotations, far too many for manual inspection. At this point, Pat and Chris are faced with two choices: spend a long time examining each strictness annotation or cope with the unpredictability of running their optimized and uninspected program that may produce dangerous behavior on certain types of input.

1.2 My Thesis

In this thesis, I demonstrate that we can dramatically reduce the number of strictness annotations generated by AUTOBAHN 1.0 using the Haskell GHC compiler's profiling tool. I have implemented an improved version of the optimizer called AUTOBAHN 2.0, which minimizes generated strictness annotations while maintaining performance of the optimized program. When evaluated on the NoFib benchmark, AUTOBAHN 2.0 reduced the number of inferred bangs by 90.2% on average, while only degrading program performance by 15.7% compared with the performance produced by AUTOBAHN 1.0. In a case study on a garbage collection simulator, AUTOBAHN 2.0 eliminated 81.8% of the recommended bangs, with the same 15.7% optimization degradation when compared with AUTOBAHN 1.0.

Apart from specifying a few additional optimization parameters, users do not need to do much extra work to use AUTOBAHN 2.0. AUTOBAHN 2.0 also does not impose any significant additional time costs on top of what AUTOBAHN 1.0 already demands. Similar to running AUTOBAHN 1.0, users will still need to run AUTOBAHN 2.0 over night as the genetic algorithm typically runs for about 100 times longer than the program being optimized. For this reason, Pat and Chris should continue to run AUTOBAHN 2.0 at the end of their deployment cycle after their program has been fully implemented. However, users will typically need to spend much less time manually inspecting the minimal strictness annotations generated by AUTOBAHN 2.0.

1.3 Lazy Evaluation

To understand the solution to Pat and Chris' dilemma, it is useful to first understand the root of their performance problem - Haskell's lazy evaluation strategy. 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 **PeytonJones89** Most of the time lazy evaluation is beneficial - it supports modularity, can improve program efficiency, and enables the use of infinite data structures **Hughes89**

While lazy evaluation reaps many benefits, it can also cause serious performance problems in both time and space when large amounts of memory are allocated to thunks **Jones94**; **Santos98**; **Ennals03** This is also why Pat and Chris' program runs much slower than anticipated. Reducing the number of thunks created at runtime is an important optimization in the GHC compiler, which uses a backward static analysis **Sergey14** to statically find expressions that the compiler can safely evaluate immediately rather than converting into thunks. Because this analysis is part of the GHC compiler, it must be conservative, eliminating only thunks that it can prove will not affect the program semantics when given any possible input. Unfortunately, this conservative analysis is not always sufficient, in which case programmers can manually insert strictness annotations such as *bang patterns* **bang** which instruct the compiler to immediately evaluate the corresponding expression regardless of whether the strictness analysis determines it is safe to do so. These manual strictness annotations can significantly improve performance **rwh** 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 **Mitchell13**

1.4 AUTOBAHN 1.0

AUTOBAHN 1.0 **autobahn-wang** is a tool that helps Haskell programmers reduce their thunk usage by automatically inferring strictness annotations. Users provide AUTOBAHN 1.0 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 1.0 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. Preserved candidates are then either mutated or combined to produce newer generations of candidate bang placements. When the budgeted optimization time is up, AUTOBAHN 1.0 returns a list of well-performing candidates, ranked by how much each candidate improves program performance. Users can then inspect the candidate bang placements and decide if they want to apply one of them to the program.

1.5 AUTOBAHN 2.0

This thesis presents AUTOBAHN 2.0, an improved version of AUTOBAHN 1.0 that aims to reduce the number of generated bangs. AUTOBAHN 2.0 introduces a *pre-search phase* and *post-search phase* that run before and after AUTOBAHN 1.0, respectively. Both phases use information from GHC's profiler to locate and eliminate ineffective bangs. GHC profiles are helpful in this regard because they show the amount of runtime and memory each location in the program is responsible for. The pre-search phase uses this information to adjust the set of files that AUTOBAHN 1.0 considers during its optimization. Specifically, this phase instructs AUTOBAHN 1.0 to optimize files that contain locations that require significant resources, skipping files that do not and potentially adding files not originally included by the users. After AUTOBAHN 1.0 runs, the post-search phase individually tests each candidate bang that falls within a costly location. Bangs in costly locations that do not significantly impact program performance are eliminated, as well as bangs that do not fall in costly locations. The system is parameterized, so users can manage the tradeoff between aggressively reducing the number of bangs and preserving performance improvements. In our experiments, we chose to aggressively reduce the number bangs, accepting some performance degradation over the level of optimization provided by AUTOBAHN 1.0.

The contribution of this paper are the following:

- We describe how to use profiling information to automatically reduce the number of bangs inferred by AUTOBAHN 1.0 while maintaining roughly the same level of optimization.
- We show that AUTOBAHN 2.0 applied to the NoFib benchmark suite reduced the number of generated bangs by 90.2% on average, while increasing the runtime of the optimized program by 15.7% over the runtime of the program optimized by AUTOBAHN 1.0 alone. We refer to this performance change as a 15.7% *optimization degradation*.
- We demonstrate that the pre-search phase removed at least one file from consideration in 21 of the benchmarks in the NoFib suite, corresponding to 35% of the programs we considered. For these programs, the pre-search phase eliminated 45 potential bang locations per 100 LOC, resulting in a mean bang reduction of 87.79% across the entire benchmark suite.
- We use a microbenchmark to show that the pre-search phase's suggestions for additional files to consider can improve AUTOBAHN 1.0's optimization results by 86.6%.
- We evaluate the post-search phase on the NoFib benchmarks, showing it can reduce the number of inferred bangs by 93.8% with a 33% optimization degradation.

- We use AUTOBAHN 2.0 in a case study to optimize the performance of `gcSimulator` **Ricci13** a garbage collector simulator. The system reduced the number of inferred bangs by 81.8% with a 15.7% optimization degradation.

Chapter 2

Background

2.1 GHC Profiling and Cost Centers

Intuitively, AUTOBAHN 1.0's genetic algorithm generates too many bangs in Pat and Chris' program because it conducts a purely random search for locations to insert bangs. To focus the search on bangs that are likely to impact performance, AUTOBAHN 2.0 makes use of the information provided by GHC's profiling system. This system allows users to better understand which locations in a program consume the most resources. The system adds annotations to the program and generates a report detailing the amount of time, memory allocations, or heap usage that is attributable to each location. To generate these profiles, users simply run their program on representative input after re-compiling it with options to request profiling. Users can choose to generate either a time and allocation or a heap profile, as well as the method by which the profiling system adds annotations. Users can manually specify annotations or invoke the profiler with the `-prof -fprof-auto` option, which automatically adds an annotation around every binding in the program that is not marked `INLINE`. In the generated profile, each annotation gives rise to a *cost center* that indicates how much time or memory were attributable to the given program point as a percentage of the whole program's resource utilization.

2.2 Genes and Chromosomes

Cost center profiling provides guidance for the otherwise random search that AUTOBAHN 1.0 performs using a genetic algorithm. In the algorithm, any program source location where a bang may be added is represented as a *gene* that can be turned *on*, corresponding to the presence of a bang, or *off*, corresponding to its absence. Since it doesn't make sense to put two bangs in one program location, there are a fixed number of possible bang locations in a given program. A *chromosome* comprises all of the genes within a file. We represent a chromosome as a fixed-length bit vector, in which each bit indicates the presence or absence of a bang at the corresponding location. Since a program is a collection of source files, it is represented as a collection of bit vectors, or chromosomes.

2.3 AUTOBAHN 1.0's Genetic Algorithm

AUTOBAHN 1.0'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 1.0 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 1.0 uses a unique number generator each time to guarantee randomness.

2.4 Optimization Coverage

AUTOBAHN 1.0 uses program source files as the unit of granularity for the set of program locations to consider when inferring bangs. By default, AUTOBAHN 1.0 optimizes all source code files in the program's directory. In other words, AUTOBAHN 1.0 by default represents all files in the directory using a list of chromosomes in which each chromosome corresponds to one file. However, Pat and Chris can specify a different set of files by manually adding or removing file paths in the AUTOBAHN 1.0 configuration. We call the set of files considered in a given run of AUTOBAHN 1.0 its *optimization coverage*. AUTOBAHN 1.0 does not infer bangs for external libraries that are imported by the program, but Pat and Chris can manually add local copies of the source code for such libraries to include them in the optimization process.

2.5 Representative Input

To run AUTOBAHN 1.0, Pat and Chris need to provide input on which to run their program. The input should be short enough for AUTOBAHN 1.0 to finish execution in a reasonable amount of time while still be long enough for it to measure noticeable time improvements if there are any. The input should also be representative of the data that Pat and Chris intend to supply to the finished program so the system optimizes the program for that kind of data. It is this ability to optimize for relevant input and not all input that gives bangs, whether added by Autobahn or by the user, the ability to outperform GHC's optimizer.

Chapter 3

AUTOBAHN 2.0

3.1 Why Too Many Bangs Are Generated

The first step towards reducing the number of inferred bangs is to identify categories of bangs and to hypothesize why each such category exists. To that end, we identify three categories of bangs:

1. a *dangerous* bang is one that can significantly degrade program performance, including causing non-termination
2. a *useless* bang neither improves nor degrades performance but is undesirable because programmers must waste valuable time reasoning about its safety
3. a *useful* bang improves program performance

An unfit chromosome performs poorly as a whole, but it can contain a mixture of dangerous, useless, and useful bangs (By definition, it must contain at least one dangerous bang). When interatively measuring performances of candidate chromosomes, AUTOBAHN 1.0 handles unfit chromosomes by discarding them entirely. It does not attempt to identify the dangerous bangs within the chromosome. Individuallly removing such bangs from consideration, rather than simply discarding the chromosome they belong in, would be useful because otherwise they can reappear in later generations as the result of a random mutation.

Fit chromosomes face a similar issue. AUTOBAHN 1.0 handles fit chromosomes by preserving the entire chromosome, without separating the useful bangs from the useless ones. This lack of discrimination is problematic for two reasons. First, we might lose useful bangs in future generations because we cannot guarantee that they will be preserved by the mutation and crossover operations of the genetic algorithm. Second, useless bangs can survive if they are grouped with useful ones. The accumulation of such useless bangs can dramatically increase the number of inferred bangs, leaving the user with a substantial reasoning task. As program source code increases in size, the number of useless bang positions also grows, increasing the likelihood the genetic algorithm will infer and preserve useless bangs that happen to live on a chromosome with useful ones.

3.2 The Solution

3.2.1 Identifying Bang Categories

Precisely identifying which bangs are useless or useful is undecidable, but we can use GHC profiling to make an educated guess. Cost centers 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. Intuitively, a bang that appears in a cost center that uses many resources, which we call a *hot spot*, is more likely to be useful because a bang-induced change in performance at the hot spot will likely significantly affect overall runtime as well. On the other hand, a bang in a cost center using few resources, which we call a *cold spot*, is likely to be useless because executing code at a cold spot occupies an insignificant portion of the overall program runtime to begin with. Leveraging that intuition, AUTOBAHN 2.0 preserves bangs that appear in hot spots while eliminating those in cold spots.

We introduce the *hotSpotThreshold* parameter to determine which cost centers to consider hot: a cost center consuming more than *hotSpotThreshold* resources is considered hot, while one consuming fewer is cold. Currently, we set *hotSpotThreshold* to 6% of the overall program runtime, although that threshold can be adjusted through user specification. As the threshold increases, AUTOBAHN 2.0 preserves fewer bangs at the risk of greater degradations in program performance.

3.2.2 AUTOBAHN 2.0's Phases

AUTOBAHN 2.0 uses GHC profiling information to reduce the number of generated bangs in two different ways, corresponding to two different phases in the optimization pipeline, which is shown in Figure 3.1. The first phase, which we call the Pre-search phase, eliminates any chromosome corresponding to a program source code file that contains only cold spots. AUTOBAHN 2.0 invokes AUTOBAHN 1.0 only on chromosomes corresponding to files that contain at least one hot spot. The pre-search phase is beneficial for two reasons. Firstly, removing chromosomes that are unlikely to contain useful bangs reduces the size of the search space with little chance of inadvertently eliminating good sets of annotations. The smaller search space is likely to make AUTOBAHN 1.0 more effective in improving performance as the optimizer will have a chance to explore more possible combinations of annotations, therefore producing higher quality bang candidates. Secondly, the optimizer will not be able to infer bangs for any of the genes in the eliminated chromosomes. Since such bangs are very likely to have been useless as they are only located in cold spots, this step is very likely to reduce the number of inferred useless bangs.

The second phase, which we call the Post-search phase, uses profiling information to categorize useless or useful bangs that occur within hot spots on chromosomes that AUTOBAHN 1.0 determines to be fit. In a world with infinite resources, we would test all combinations of such bangs and select the best one because the effect of combining bangs is hard to predict. Unfortunately, the exponential search space is too large to explore exhaustively. Instead, we individually turn off one such bang at a time, measure the resulting performance, and keep only those that significantly impact performance. The number of such bangs is sufficiently small that we can test each in turn because the number of bangs in hot spots in a program is relatively small.

In the rest of this section, we explain AUTOBAHN 2.0's optimization pipeline in more detail.

3.3 Pre-search Profiling

Just as manually reasoning about bang placement can be difficult, so too reasoning about which files to optimize can be challenging. AUTOBAHN 2.0 makes this process easier by leveraging information in GHC profiles. To that end, the pre-search

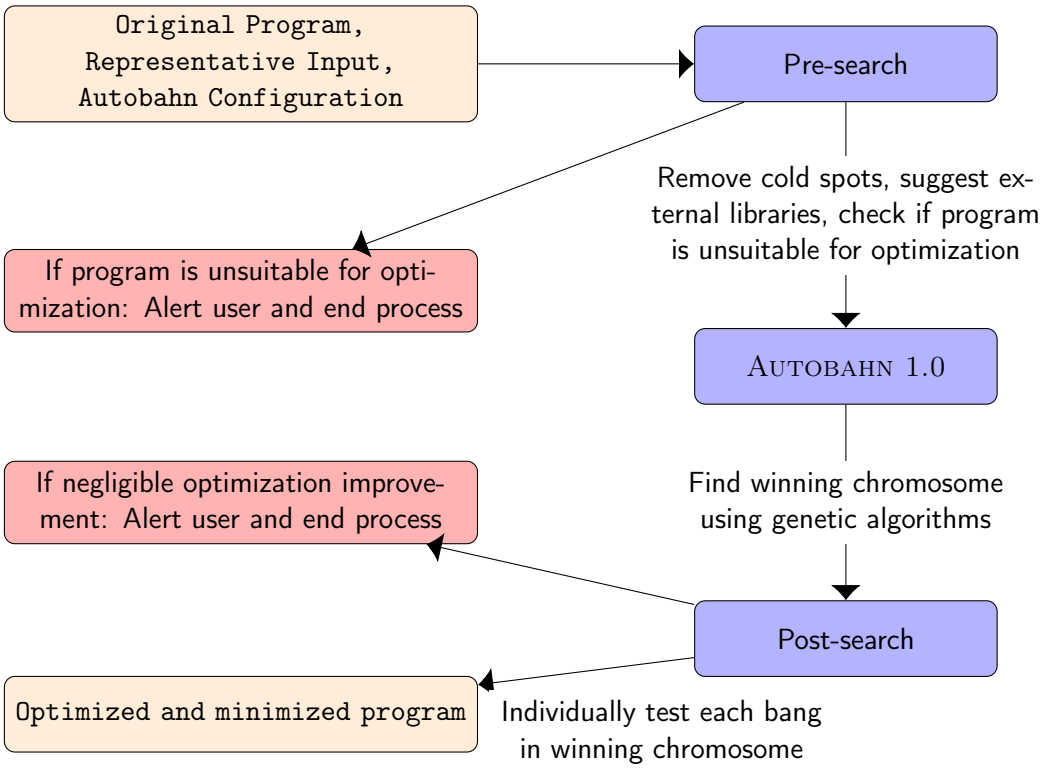

FIGURE 3.1: AUTOBAHN 2.0 optimization pipeline.

phase begins by generating a GHC time and allocation profile for the unoptimized program by running it on user-provided representative input. AUTOBAHN 2.0 then sets the optimization coverage for AUTOBAHN 1.0 to be source files that contain at least one hot spot. In addition, AUTOBAHN 2.0 identifies library files that contain hot spots and suggests to users that they add local copies so the library source files may be optimized as well. AUTOBAHN 1.0 then optimizes the program as usual, except using a more targeted set of files/chromosomes. Note that this phase can both expand the search space, if a user chooses to add the identified library files that contain hotspots, as well as reduce it, by identifying program source files with no hot spots.

Pre-search profiling offers three important benefits. First, it can greatly reduce the number of bangs AUTOBAHN 1.0 suggests by limiting the search space to those program files that have a chance of significantly impacting performance. This focusing reduces the possibility of generating useless or dangerous bangs by eliminating them from consideration and increases the chances of generating useful ones by allowing more of the relevant search space to be explored within a given time budget. Second, if a hot spot is located in an external library file, the pre-search phase can identify the relevant files so they can be included in the optimization process. Third, it identifies programs that are potentially unsuitable for AUTOBAHN 1.0 optimization. If a program contains a large number of cost centers that all contribute minimally to program runtime, there may not be any way to substantially improve program performance by adding bang annotations. If the pre-search phase concludes that a program only contains cold spots, it will alert the user and abort, saving the time and effort of running the remaining phases, which are unlikely to identify significant performance improvements.

It is worth noting that although the pre-search phase can reduce the size of the

search space, it does not reduce AUTOBAHN 1.0's runtime. AUTOBAHN 1.0 will always search until exhausting its time budget. The reduced search space does allow AUTOBAHN 1.0 to explore the space more thoroughly, statistically speaking, potentially allowing it to find better results within the fixed time budget.

3.4 Post-search Bang Elimination

After AUTOBAHN 1.0 explores the search space and suggests a winning set of chromosomes, AUTOBAHN 2.0 uses GHC profiling information to eliminate any bangs on those chromosomes that don't significantly contribute to program performance. To that end, the post-search phase decides that any gene that is off in the winning chromosomes should not have a bang in the final recommendation. It removes these genes from further consideration. The post-search phase then maps each remaining gene, which must be turned on, to its corresponding cost center. It turns off all genes that do not fall within hot spots, deciding not to recommend bangs for the corresponding locations on the theory that such bangs are likely to be useless. It then removes these genes from further consideration.

The remaining genes are the interesting ones in that they both contain a bang and fall within a hot spot. These genes require further testing because they may still be useless, failing to improve program performance even though they fall within a hot spot. There is usually a sufficiently small number of remaining genes that are both turned on and within a hot spot that it is possible to measure their impact one by one. Specifically, the post-search phase tests each such gene in turn, turning it off while keeping all the remaining bangs on. It then runs the resulting program and compares its performance to that of the winning chromosomes as determined by AUTOBAHN 1.0. If the absence of the bang slows down the program by the value of the *absenceImpact* threshold parameter, the post-search phase deems the bang useful and decides to keep it. Otherwise, the bang is deemed useless and is discarded. The *absenceImpact* threshold is adjustable; we set it to 6%. The post-search phase repeats this process for every bang that is both in the winning chromosomes and in a hot spot. The minimization result is the combination of all the surviving bangs.

It is possible that there are no surviving bangs at the end of testing, and that occurs when there are no hot spot bangs remaining after all the cold spot bangs are eliminated. Another possibility is that the combination of hot spot bangs prior to testing slows down the program runtime to equal to or more than the original runtime, in which case the pre-search phase will remove all bangs and return the original program.

3.5 Representative Input

Just as with AUTOBAHN 1.0, the quality of the representative input impacts the quality of AUTOBAHN 2.0's performance because different inputs may generate wildly different results in GHC profiles. Therefore, users must carefully choose input that well represents typical types of data the program might receive.

Chapter 4

Implementation

4.1 Program Architecture

AUTOBAHN 2.0 wraps AUTOBAHN 1.0 with the pre-search and post-search phases as shown in Figure 3.1. Prior to running the genetic algorithm, the pre-search phase invokes GHC's profiler on the unoptimized version of the user's program with the `-prof -fprof-auto` option to auto-generate cost centers around every non-inlined program binding. It uses the supplied representative data as input to generate the time and allocation profile. This baseline profile serves as the single point of reference for the rest of AUTOBAHN 2.0 to refer to for hot spot information throughout the entire optimization pipeline. Depending on the where the hot spots fall according to the profiler, the program's optimization coverage will either be automatically reduced or manually expanded by the user.

Then AUTOBAHN 2.0 reuses the existing implementation of AUTOBAHN 1.0 **autobahn-wang** to identify a winning set of chromosomes. AUTOBAHN 1.0 uses the *haskell-src-exts* **langexts** parser to parse source files and to identify genes. It then applies a genetic algorithm, implemented using the `GA` Haskell library **genetic** with a fitness function based on measured runtime performance, to search for the best performing chromosomes.

The post-search phase eliminates bangs from the best-performing chromosomes returned from AUTOBAHN 1.0. It does so by first turning off all bangs that lie within a cold spot before re-running the program once for each gene that both lies in a hot spot and is turned on in the winning chromosomes. If a bang's absence negatively impacts performance by at least 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 bang does not meet the *absenceImpact* threshold, it is removed for future rounds of testing and will not appear in the final combination. Similar to AUTOBAHN 1.0, the post-search phase uses the *haskell-src-exts* library to parse the relevant source files, set all the bangs appropriately, and then pretty-print the modified source code to then be compiled by GHC and run on the representative input.

Note that this process does not require re-running the original profile. Instead, we reuse the profiling information calculated during the pre-search phase. Bangs in hot spots are tested in order of decreasing costs. While we recognize that the order in which we test the bangs may affect the performance, it is too time consuming to test the bangs in all possible orders. For simplicity, we chose to test each bang once in order of decreasing costs.

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 chromosomes that improved program runtime by at least the *overallThreshold* parameter using genetic

Term	Type	Description
diversityRate	float	probability with which each gene in seed is mutated to form initial population
numGenerations	int	number of generations to run algorithm
populationSize	int	number of chromosomes in each population
archiveSize	int	number of chromosomes to promote to next generation unchanged
mutateRate	float	percentage of the new population generated by mutation
mutateProb	float	probability with which each gene in a chromosome selected for mutation is changed
crossRate	float	percentage of the new population generated by crossover
absenceImpact	float	threshold for evaluating whether the absence of a bang has a runtime impact significiant enough to preserve the bang
hotSpotThreshold	float	threshold for evaluating which cost center qualifies as a hot spot based on its cost
profMetric	string	metric to parse GHC profiles and evaluate hot spots with. (can be either "MEM" for memory allocations or "RT" for runtime)

TABLE 4.1: AUTOBAHN 2.0 parameters

algorithms, then the post-search phase will return the unoptimized program. If AUTOBAHN 2.0 failed to find minimized bang placements that preserve program runtime, the post-search phase will also return the unoptimized program. We base this choice on the theory that the relatively insignificant performance improvement indicates users are better off keeping the original program rather than having to reason about the safety of the inferred bangs. Currently, we set the both the *overallThreshold* and *absenceImpact* value to 6%, but users may change it.

4.2 Running AUTOBAHN 2.0

Users run AUTOBAHN 2.0 the same way as they would run AUTOBAHN 1.0. They provide a copy of their program source code, representative input, and an optional configuration file. The pre-search and post-search phases typically require a negligible amount of time to run compared with the time required to run AUTOBAHN 1.0.

4.2.1 Choosing Configuration Parameters

As shown in Figure 4.1, AUTOBAHN 2.0 requires a few additional configuration parameters than what AUTOBAHN 1.0 already requires. Choosing parameters that maximize performance optimization and minimize bangs is difficult and the ideal

threshold will often vary from program to program. The *hotSpotThreshold* decides whether a cost center is costly enough to be considered a hot spot. Higher *hotSpotThreshold* thresholds will result in fewer hot spots and therefore fewer bangs, at the risk of lower performance improvement due to too many cost centers being eliminated. The *absenceImpact* determines whether a bang's contribution to performance impact is large enough to be preserved. Higher *absenceImpact* thresholds will similarly result in fewer preserved bangs and therefore less manual inspection time, at the risk of lower performance improvement due to too many bangs being eliminated. The *profMetric* specifies which metric to evaluate cost centers on, either using run time or memory allocations. In our experiments, we found that depending on the program, one profiling metric could be ideal over the other or not make a measurable difference at all.

4.2.2 AUTOBAHN 2.0 Output

If AUTOBAHN 2.0 completes successfully, users can find the resulting source code with the minimized bang annotations in the same project directory as they would find the usual AUTOBAHN 1.0 `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 warns the user and halts the optimization process. If external libraries could be added to the optimization coverage to potentially improve the results, the pre-search phase alerts the user and then continues to optimize as normal.

4.3 Source Locations

AUTOBAHN 1.0 uses a bit vector to represent the genes in a chromosome, with each bit corresponding to a possible bang location in a source file. In AUTOBAHN 2.0, we modified this representation so that each bit also indicates the cost center associated with the possible bang location. Cost centers are uniquely identified by the corresponding source locations in source files. Thus, we mapped each bit 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

Both AUTOBAHN 1.0 and AUTOBAHN 2.0 use the *haskell-src-exts* parser to add and remove bangs. Unfortunately, the version of *haskell-src-exts* parser that we use incorrectly identifies the left-hand side of bindings within instance declarations as potential locations to place bangs. For that reason, AUTOBAHN 1.0 did not optimize files that contain instance declarations. To eliminate this restriction, we instead removed any randomly generated illegal bangs prior to each round of fitness evaluation in the genetic algorithm. The rest of the genetic search 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 the 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 the AST to construct the

validity vector. As we traversed the AST, 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 `false` in the validity vector. All other legal bang locations are marked as `true`.

With this approach, AUTOBAHN 2.0 successfully avoids inserting bangs into illegal locations. As a result, AUTOBAHN 2.0 can now optimize files that include instance declarations, which was not previously possible.

Chapter 5

Evaluation

5.1 Experimental Setup

All versions of Autobahn were compiled using GHC version `8.0.2`. The NoFib benchmarks were compiled with GHC version `8.0.2` using NoFib's default flags, the flag `-XBangPatterns`, and the NoFib flag that enables profiling. We discarded the benchmarks in the NoFib suite that failed to compile or run out of the box. We also excluded the benchmarks that AUTOBAHN 1.0 refuses to optimize because they already have very fast runtimes. That left 60 benchmarks on which we could run experiments, listed in Figure 5.1.

5.1.1 Runtime Performance Ratio

To study runtime performance results, we report what we call the *runtime performance ratio*, which is the ratio of the optimized program's runtime to the runtime of the original program. A runtime performance ratio of 1 indicates that the optimized version of the program has the same runtime as the original version and is therefore *unimproved*. A runtime performance ratio of 2 is a sentinel value indicating that the "optimized" version of the program is *non-terminating*, by which we mean it runs for more than a timeout threshold longer than the unoptimized program.

5.1.2 Successes and Fails

A ratio that reflects a performance improvement of more than *absenceImpact* is considered successful because such a ratio indicates the optimized program performs substantially better than the original, while ratios that improve less than *absenceImpact* are failures. Failures can occur either as a result of AUTOBAHN 1.0 failing to find performance improving bang placements or AUTOBAHN 2.0 failing to eliminate bangs without significantly worsening runtime performance. Failures have the effect of leaving the program unchanged as the proposed annotations are discarded in favor of the original (or in favor of the AUTOBAHN 1.0 optimized version, if the failure is due to AUTOBAHN 2.0). When we study the results of a particular optimization system on a particular benchmark, we also categorize the results into one of three groups: complete success, partial success, and complete failure. A benchmark completely succeeds if all of its 10 runs are successful, completely fails if all of its runs are failures, and partially succeeds if there is a mixture of these outcomes.

5.1.3 Approach

To calculate the performance of a particular benchmark using a particular optimization system (*e.g.*, AUTOBAHN 2.0 or a restricted version using only a subset AUTOBAHN 2.0's phases), we ran the system under test on the benchmark 10 times to

account for random fluctuations. We also test the performance of the benchmark on AUTOBAHN 2.0 using both runtime and memory allocations as the profiling metric. We scored each run by its runtime performance ratio. Note that each run of the benchmark can produce a different result not only because of variations in machine load but also because the search process is randomized. Failed runs can be further categorized into runs in which AUTOBAHN 1.0 fails and runs in which the post-search phase of AUTOBAHN 2.0 fails. If a benchmark failed at the AUTOBAHN 1.0 stage, then we say that its runtime performance ratio for that run is 1 and it recommended 0 bangs. If a benchmark succeeded at the AUTOBAHN 1.0 stage but failed in the post-search phase, then we say its runtime performance ratio is 1 (because we discard the optimization results) and the number of inferred bangs is 0 bangs, because we discard the optimization annotations. Finally, we computed the average of the runtime performance ratios and the average of the number of recommended bangs.

Program	Loc	Files	Genes
wheel-sieve1	41	1	38
wheel-sieve2	47	1	47
callback002	41	1	3
rfib	12	1	5
threads007	16	1	5
x2n1	35	1	6
threads001	15	1	8
callback001	41	1	9
tak	16	1	9
mutstore2	22	2	10
spellcheck	13	1	10
threads003	20	1	11
mutstore1	25	2	12
primes	18	1	12
queens	19	1	18
bernouilli	40	1	19
kahan	58	1	23
exp38	93	1	24
pidigits	22	1	27
integrate	43	1	28
cryptarithm1	164	1	33
fasta	58	1	44
integer	68	1	47
life	53	1	48
rsa	74	2	50
binary-trees	74	1	51
maillist	178	1	52
gcd	60	1	57
cryptarithm2	128	1	60
lcss	60	1	71

Program	Loc	Files	Genes
atom	188	1	74
paraffins	91	1	75
calendar	140	1	92
awards	115	2	99
fish	128	1	102
puzzle	170	1	103
treejoin	121	1	119
eliza	267	1	138
sorting	131	2	196
cichelli	195	4	205
cse	464	2	222
pic	527	9	235
clausify	184	1	246
minimax	238	6	299
boyer2	723	5	302
expert	525	6	424
hidden	507	14	430
gamteb	701	13	458
prolog	643	9	514
infer	600	16	586
fem	1286	17	655
scs	585	7	770
simple	1129	1	845
reptile	1522	13	895
symalg	1024	11	1137
gg	812	9	1192
cacheprof	2151	3	1228
fulsom	1392	13	1433
fluid	2401	18	1688
anna	9561	32	7709

TABLE 5.1: Statistics for the NoFib benchmarks

5.2 Pre-search Search Space Reduction

To assess the impact of the pre-search phase, we study the number of genes that the phase eliminated from (or added to) consideration by AUTOBAHN 1.0. We compare the number of bangs that AUTOBAHN 1.0 inferred when run with and without the pre-search phase, calling the optimizer comprised of the pre-search phase plus AUTOBAHN 1.0 the *Pre-search optimizer.* We note in how many of the runs the Pre-search optimizer failed. Finally, we compare the runtime performance ratio for AUTOBAHN 1.0 with the ratio for the Pre-search optimizer. To compute this data, we ran both the Pre-search optimizer and AUTOBAHN 1.0 on the 60 programs from NoFib benchmark suite. We optimized all runs using runtime as the profiling metric only, and we set both the *hotSpotThreshold* and *absenceImpact* thresholds to 6%.

The pre-search phase eliminated at least one file in 21 out of the 60 benchmark programs. The remaining 39 benchmarks fall into one of two categories. The first category consists of 33 benchmarks that did not have any files eliminated by the pre-search phase. The second category comprises 6 benchmarks that the pre-search phase determined were not suitable for optimization because they had no significant hot spots. Since AUTOBAHN 1.0 and the Pre-search optimizer handle these 39 benchmarks identically, we exclude them from the graphs reporting on the results of the pre-search phase. We discuss the second category of benchmarks in Section 5.3. None of the 60 programs in the benchmark suite had an external hot spot not in the original optimization coverage.

Figure 5.1 shows the results from the 21 benchmarks that had at least one file eliminated during the pre-search phase. The column `Eliminated Genes` gives the number of potential bang locations that were eliminated before AUTOBAHN 1.0 ran. The `Original Bangs` column records the number of bangs in the original program, most of which were zero. The `Failed runs` column gives the fraction of the 10 runs on which the Pre-search optimizer failed. Finally, the `AUTOBAHN 1.0 Bangs` and `Pre-search Bangs` columns give the number of recommended bangs by the two systems, respectively. On average, the pre-search phase eliminated 45 genes per 100 LOC across the 60 programs in the benchmark suite before invoking AUTOBAHN 1.0. The Pre-search optimizer recommended 87.79% fewer bangs than AUTOBAHN 1.0.

The `anna`, `expert`, and `symalg` benchmarks are particularly interesting because AUTOBAHN 1.0 consistently failed to find winning chromosomes for them, and so it did not generate any bangs. However, after reducing the size of the search space, the Pre-search optimizer was able to find meaningful bangs for them.

Figure 5.2 shows the corresponding runtime performance ratios produced by AUTOBAHN 1.0 and the Pre-search optimizer on the same 21 benchmarks. The graph shows that even when a large number of genes are eliminated prior to running the genetic algorithm, the optimizer is still able to find useful bangs that result in similar runtime improvement. This data confirms that the eliminated genes were on the whole not useful. On average on these 21 programs, the Pre-search optimizer returns runtime performance ratios of 0.69 compared to 0.75 for AUTOBAHN 1.0, which is an optimization *improvement*.

5.3 Pre-search File Elimination

The preopt phase identifies 6 benchmarks among the 60 in our suite as unsuitable for optimization when the *hotSpotThreshold* threshold is set to 6%: `awards`, `callback001`,

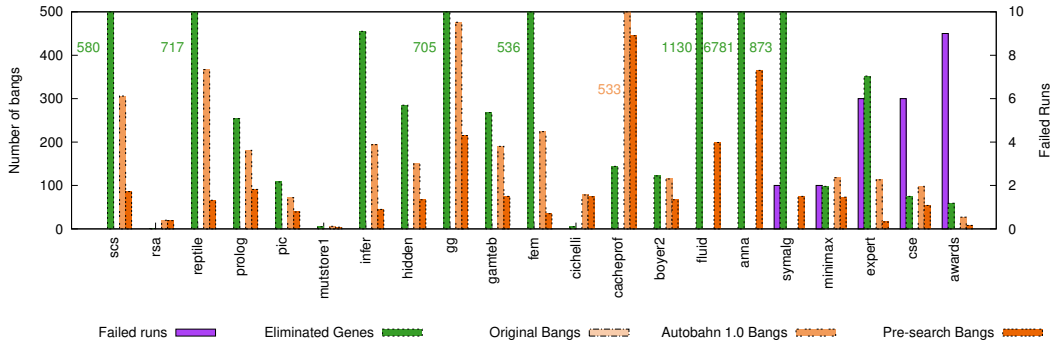

FIGURE 5.1: Number of bangs generated by AUTOBAHN 1.0 vs. Pre-search optimizer across 21 benchmarks that had at least one file eliminated during the pre-search phase. Columns that exceed the maximum axis value are labeled with their actual values. The benchmarks are sorted in increasing order of number of failed runs for the Pre-search optimizer.

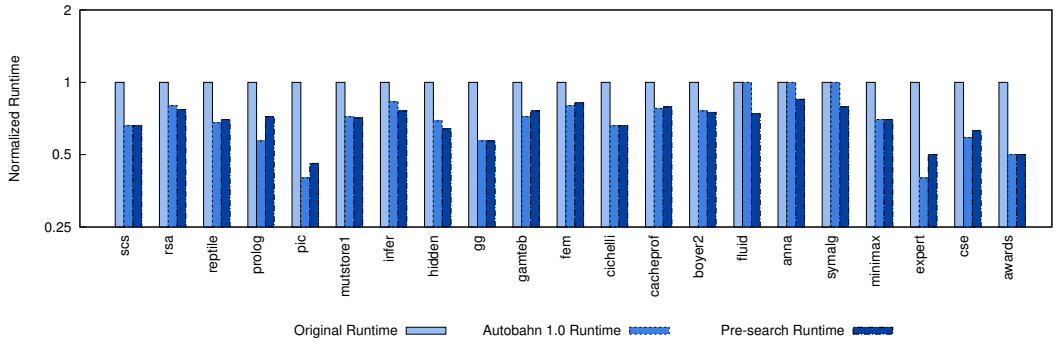

FIGURE 5.2: Performance runtime ratios of AUTOBAHN 1.0 vs. Pre-search optimizer across 21 benchmarks that had at least one file eliminated during the pre-search phase. The x-axis is on a base-2 log scale. The benchmarks appear in the same order as in Figure 5.1.

`callback002`, `mutstore2`, `sorting`, and `threads007`. As expected, when attempting to optimize `awards`, `sorting`, and `threads007`, AUTOBAHN 1.0 consistently fails, returning the unimproved runtime code. However, AUTOBAHN 1.0 was able to successfully optimize the other programs. Through inspection, we concluded that `callback002` would have benefited from a lower *hotSpotThreshold* threshold as its most costly hot spot takes up 3.9% of the program runtime. Both `callback001` and `threads007` would have 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. The `mutstore2` program'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

To demonstrate the effectiveness of using the pre-search phase to expand Autobahn's coverage for improved optimization results, we tested our approach on the `sumList` microbenchmark. We created `sumList` to simulate the scenario in which a programmer references code from an external library or external file that contains hot spots but is not within the current optimization coverage.

The `sumList` program's `Main.hs` file contains only one function: a main function that constructs a list of integers from 1 to 1,000,000 and then calculates the sum of all integers in the list using an external *sum* function located in `Sum.hs`. Users may decide to set the optimization coverage to [`Main.hs`], because they are interested in making the main program run faster. However, as demonstrated in Figure 5.3, AUTOBAHN 1.0 was only able to improve program performance by 3%, even when it was able to exhaustively search the possible bang locations in the 6 lines of code in the `main` function. Upon inspecting the results, users may be mistaken in believing that their program runtime cannot be improved further.

But there are indeed other opportunities to speed up *sumList* located in places that the users did not think about. If the users were to rerun the optimization using the pre-search phase, then GHC's time and allocation profile indicates that the largest *hotSpotThreshold* was 9.5% and located in lines 7 to 8 in the *sum* function in `Sum.hs`. The *sum* function is entirely lazy and did not immediately compute the sum of each integer as it recursed through the list. In the resulting log file, the pre-search phase suggests adding `Sum.hs` to the optimization coverage. After expanding the coverage and re-running the optimization, the resulting `sumList` ran at only 13% of the original runtime, a dramatic improvement.

Although the `sumList` example is short and synthetic, it shows the larger potential for users to obtain much better optimization results when running the pre-search phase in conjunction with AUTOBAHN 1.0. Programmers often build upon each other's code and use external functions that they may not be entirely familiar with or did not consider optimizing. The pre-search phase can identify valuable missed opportunities. Of course, the addition of more files to optimize means that more bangs might be generated. It is up to the user to decide if they want to add the suggested files for better optimization results at the risk of needing to inspect more bangs.

Version	Coverage	Normalized Run-time	No.Bangs
Original	N/A	1	0
AUTOBAHN 1.0	[`Main.hs`]	0.97	2
Pre-optimization	[`Main.hs`, `Sum.hs`]	0.13	4

FIGURE 5.3: Results for `sumList` microbenchmark.

5.5 Post-search Bang Reduction

To assess the effectiveness of the post-search phase, we compare the results of running AUTOBAHN 1.0 with running the *Post-search optimizer* comprised of AUTOBAHN 1.0 followed by the post-search phase. Figures 5.4 and 5.5 give the number of bangs and the runtime performance ratios, respectively, for the 21 benchmarks that AUTOBAHN 2.0 successfully optimized on all 10 runs. Figures 5.6 and 5.7 give the number of bangs and the runtime performance ratios, respectively, for the 28 benchmarks on which AUTOBAHN 2.0 was partially successful, sorted in increasing order of AUTOBAHN 2.0's failure rate. We do not show the results from the remaining 12 benchmarks that AUTOBAHN 2.0 failed to optimize on every run since the numbers for those benchmarks would be unchanged from the original program. Figure 5.8 shows how frequently a benchmark failed at the AUTOBAHN 1.0 stage vs. how frequently it failed at the post-search phase. The graph shows that the majority of Post-search optimizer fails are caused by failures at the AUTOBAHN 1.0 stage, which produces a poorly performing set of bangs for the post-search phase to minimize. Therefore, benchmarks that failed at the AUTOBAHN 1.0 stage are highly likely to fail during the post-search phase as well.

Figures 5.4 and 5.6 show that the number of bangs eliminated by the Post-search optimizer is quite significant: on average 93.8%. Figures 5.5 and 5.7 show the corresponding runtime performance ratios of each optimized program. In most benchmarks, the Post-search optimizer does a little worse than AUTOBAHN 1.0, on average 33% worse, because the post-search phase only preserves bangs that affect program runtime by at least the *absenceImpact* threshold (6% in these experiments). If users want to maintain higher levels of optimization, 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.

Looking at these results in more detail, the `callback001` and `pic` benchmarks are good examples of when a program has no cost centers that meet the *hotSpotThreshold* threshold, so the post-search phase eliminated all their bangs. In these cases, a lower threshold would have helped discover useful bangs that are located in the relatively colder locations. The data for `anna` and `fluid` show that while AUTOBAHN 1.0 found bangs that triggered a non-terminating runtime code, post-search bang elimination was able to eliminate the dangerous bangs that caused the bad behavior and instead produce successful optimization results.

5.6 AUTOBAHN 2.0: Combining pre-search and post-search

To evaluate the overall effectiveness of AUTOBAHN 2.0, we ran the complete tool chain on our NoFib benchmarks 10 times. Figures 5.9, 5.10, 5.11, and 5.12 presents the numbers of recommended bangs and the runtime performance ratios on the 52 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using runtime

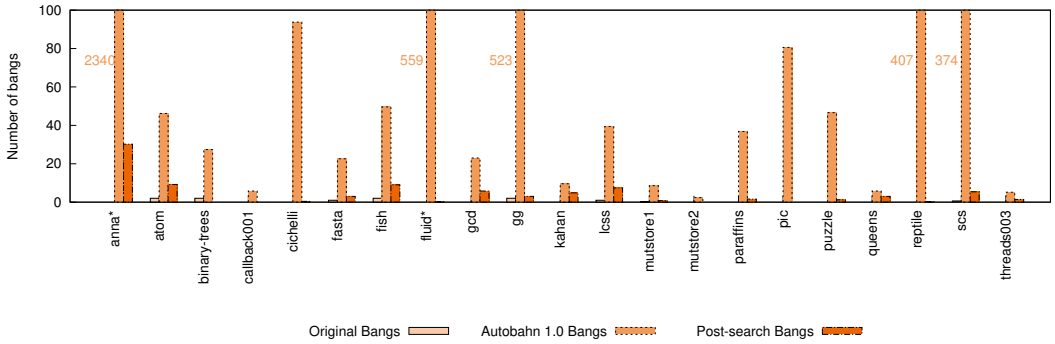

FIGURE 5.4: Number of bangs generated by AUTOBAHN 1.0 vs. Post-search optimizer across 21 benchmarks that AUTOBAHN 2.0 successfully optimized every time. Columns that exceed the maximum axis value are labeled with their actual values.

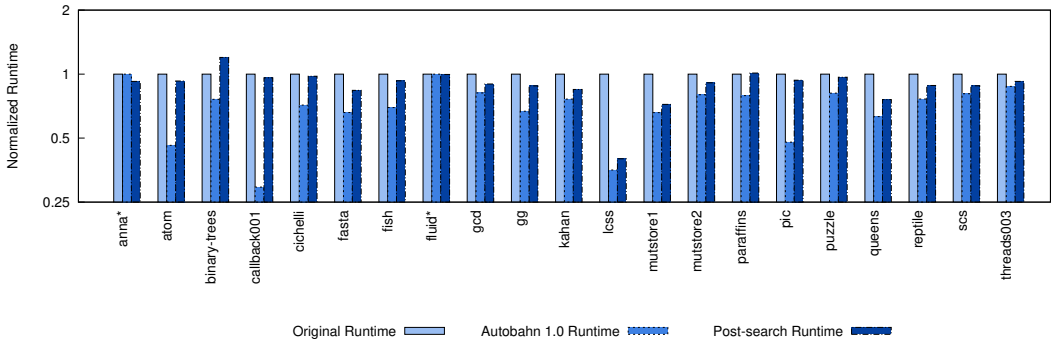

FIGURE 5.5: Performance runtime ratios of AUTOBAHN 1.0 vs. Post-search optimizer across 21 benchmarks that AUTOBAHN 2.0 successfully optimized every time. The x-axis is on a base-2 log scale. For the *-ed benchmarks, AUTOBAHN 1.0 generated non-terminating optimization results, but the post-search phase removed the dangerous bangs and succeeded in optimizing the program.

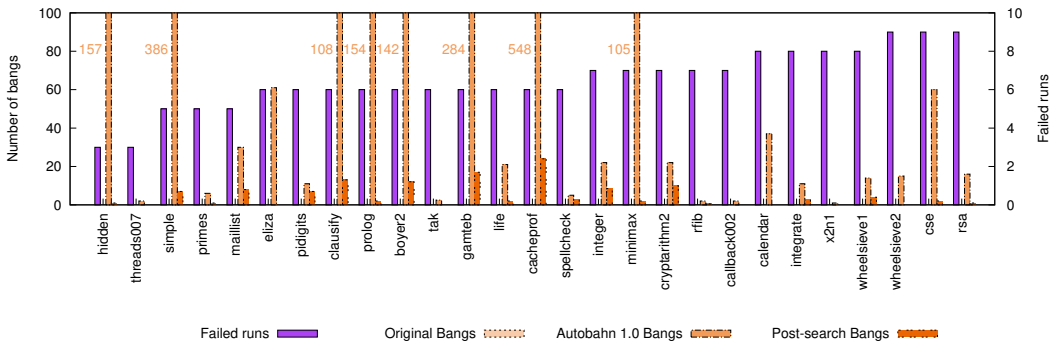

FIGURE 5.6: Number of bangs generated by AUTOBAHN 1.0 vs. Post-search optimizer across 27 benchmarks that AUTOBAHN 2.0 successfully optimized on some runs. Failure rate out of 10 runs is shown. Columns that exceed the maximum axis value are labeled with their actual values. The benchmarks are sorted in increasing order of AUTOBAHN 2.0 failure rate.

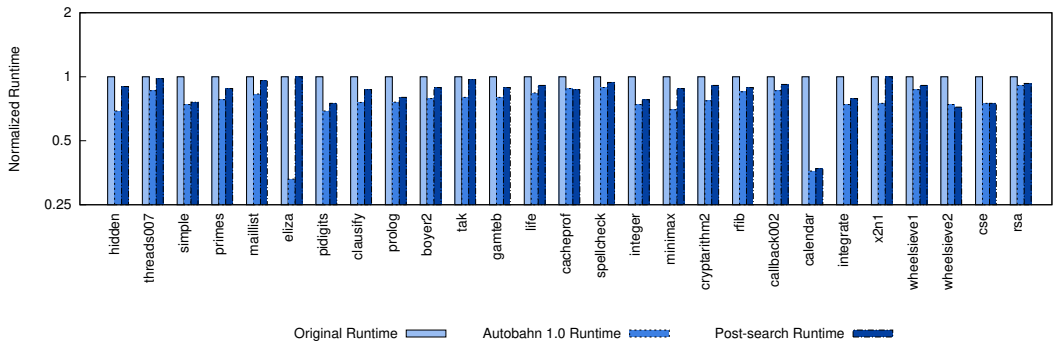

FIGURE 5.7: Performance runtime ratios of AUTOBAHN 1.0 vs. Post-search optimizer across 27 benchmarks that AUTOBAHN 2.0 successfully optimized on some runs. The x-axis is on a base-2 log scale.

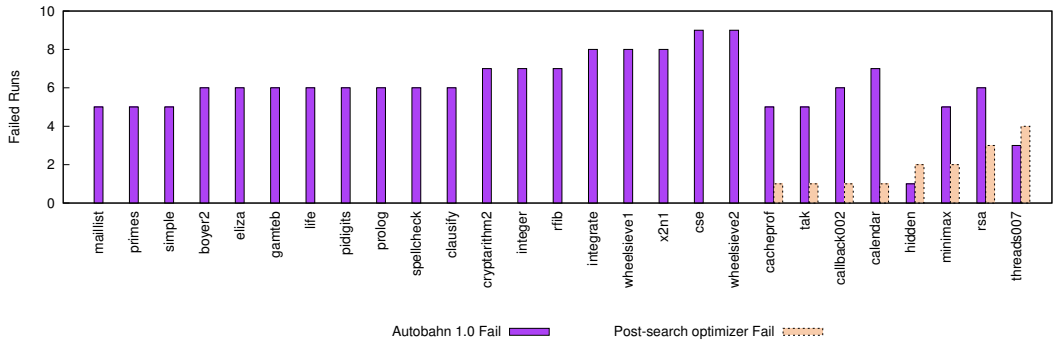

FIGURE 5.8: Frequency of failures attributable to AUTOBAHN 1.0 vs the post-search phase across 27 benchmarks on which optimization sometimes succeeded. Benchmarks are sorted in increasing order of Post-search optimizer failure rate.

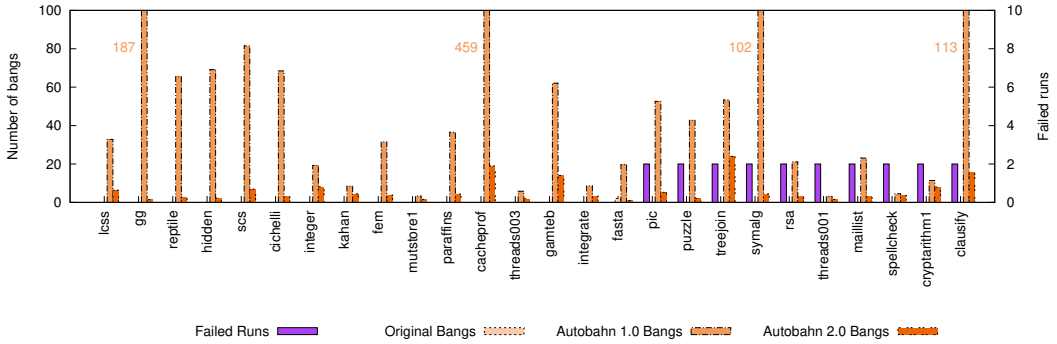

FIGURE 5.9: Number of bangs generated by AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the first 26 of 52 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using runtime as the profiling metric. Failure rate out of 10 runs is shown. Benchmarks are sorted in increasing order of failure rate. Columns that exceed the maximum axis value are labeled with their actual values.

as the profiling metric. All benchmarks are sorted in increasing order of AUTOBAHN 2.0 failure rate.

Out of those 60 programs, the pre-search phase eliminated 5 because they had no hot spots. On another 3 benchmarks, neither AUTOBAHN 1.0 nor AUTOBAHN 2.0 successfully found an optimization on any run. It is interesting to note that only these 8 benchmarks consistently failed under AUTOBAHN 2.0, while 12 consistently failed under the Post-search optimizer. This observation shows that the pre-search phase successfully increased the effectiveness of the later optimization phases.

Overall, AUTOBAHN 2.0 reduced the number of bangs generated by 90.2% with an optimization degradation of 15.7%. We split the data into two graphs of 26 benchmarks each for bang counts (Figures 5.9 and 5.11) and for runtime performance ratios (Figures 5.10 and 5.12) for legibility. Finally, Figure 5.13 shows how frequently a benchmark failed at the AUTOBAHN 1.0 stage versus at the end of AUTOBAHN 2.0. As with the post-search optimizer, the majority of runs failed at the AUTOBAHN 1.0 stage.

While most benchmarks consistently showed significant bang reduction with minimal optimization degradation under AUTOBAHN 2.0, a few benchmarks stand out. The `expert` and `calendar` benchmark not only had bangs reduced by 79.41% and 97.63% respectively, but also experienced performance *improvements* of 27.60% and 14.82% respectively. It is worth noting that both benchmarks failed more times than they succeeded, so such results are not guaranteed to be replicable in every run. The `atom` benchmark is also noteworthy because the post-search phase eliminated all bangs generated by AUTOBAHN 1.0, yet it still had a runtime performance ratio of 0.78. This finding suggests that `atom`'s original runtime fluctuates by a significant amount on its own. It also suggests that `atom`'s overall performance improvement is achieved through the accumulation of speedups at many relatively cold cost centers, so lowering AUTOBAHN 2.0's *hotSpotThreshold* might result in a better performance improvement.

5.6.1 Runtime vs. memory allocations as profiling metric

To evaluate whether there is a measurable performance difference in using memory allocations over runtime as the profiling metric, we also ran AUTOBAHN 2.0 on the 60 benchmarks 4 times (NOTE: will change this to 10 times when I finish running all

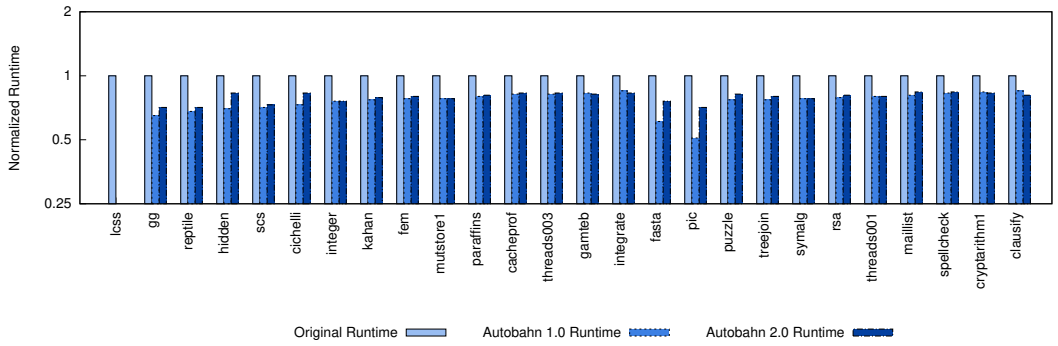

FIGURE 5.10: Runtime performance ratios of AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the first 26 of 52 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using runtime as the profiling metric. The x-axis is on a base-2 log scale.

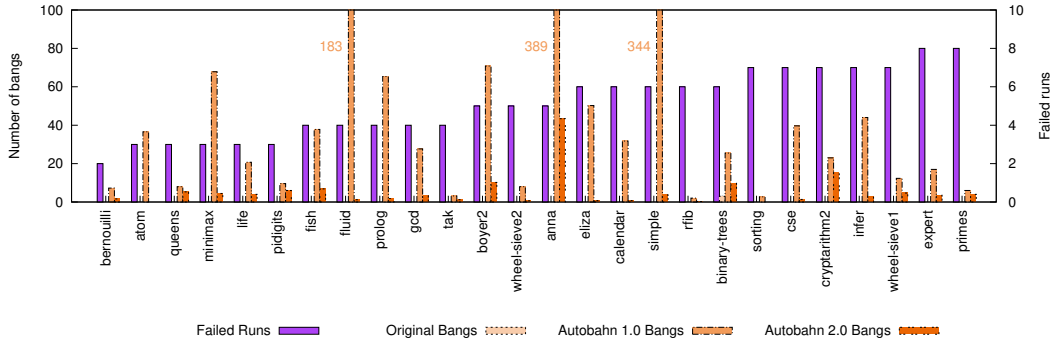

FIGURE 5.11: Number of bangs generated by AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the remaining 26 of 52 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using runtime as the profiling metric. Failure rate out of 10 runs is shown. Benchmarks are sorted in increasing order of failure rate. Columns that exceed the maximum axis value are labeled with their actual values.

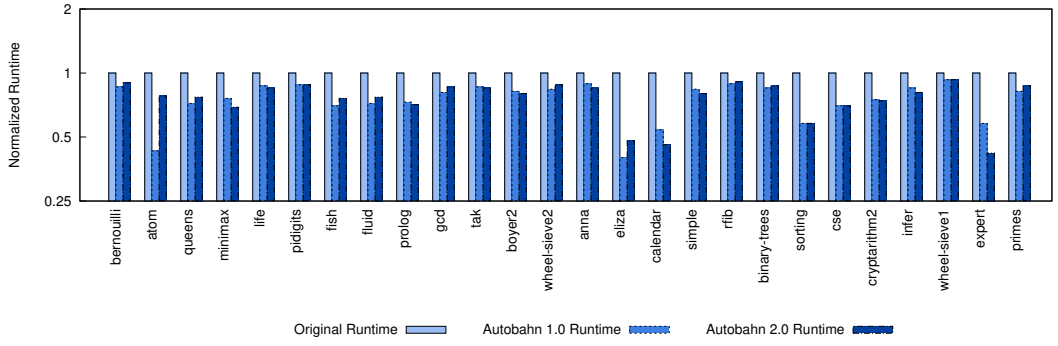

FIGURE 5.12: Runtime performance ratios of AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the remaining 26 of 52 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using runtime as the profiling metric. The x-axis is on a base-2 log scale.

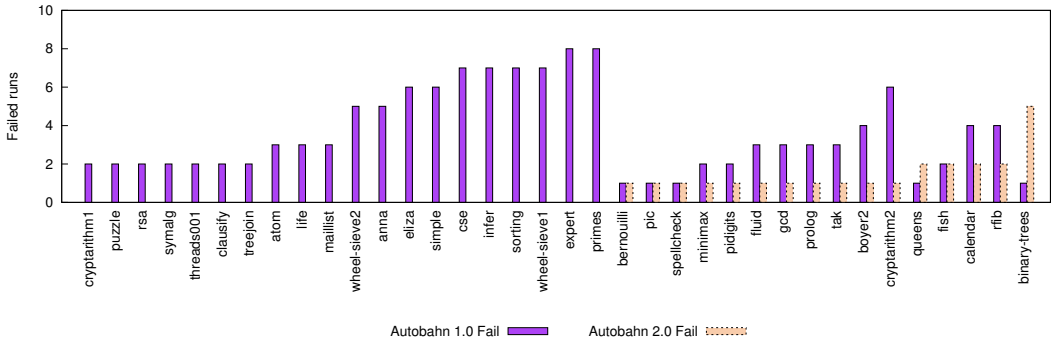

FIGURE 5.13: Frequency of failure attributable to AUTOBAHN 1.0 versus AUTOBAHN 2.0 across the 36 benchmarks that AUTOBAHN 2.0 sometimes successfully optimized using runtime as the profiling metric. Benchmarks are sorted in increasing order of AUTOBAHN 2.0 failure rate.

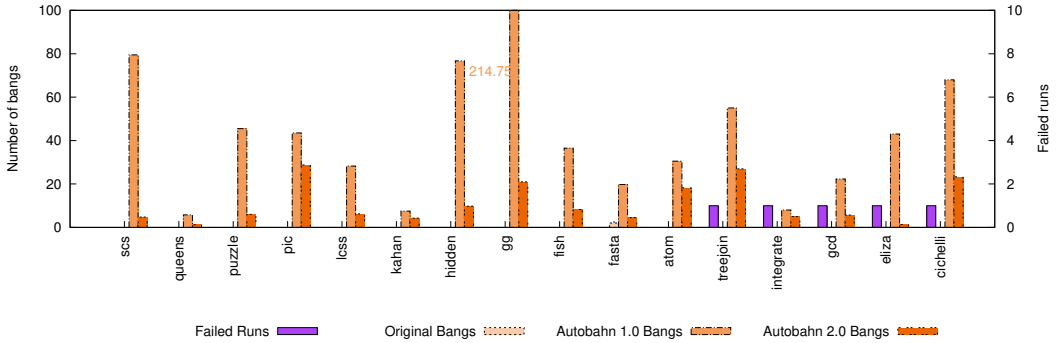

FIGURE 5.14: Number of bangs generated by AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the first 16 of 32 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using memory allocations as the profiling metric. Failure rate out of 10 runs is shown. Benchmarks are sorted in increasing order of failure rate. Columns that exceed the maximum axis value are labeled with their actual values.

of them) using memory allocations as the profiling metric. Figures 5.14, 5.15, 5.16, and 5.17 presents the numbers of recommended bangs and the runtime performance ratios on the 27 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using memory allocations as the profiling metric.

5.7 Case Study: AUTOBAHN 2.0 On `gcSimulator`

As a case study, we ran AUTOBAHN 2.0 on `gcSimulator`, which is a garbage collection simulator that consumes program execution traces produced by the Elephant Tracks **Ricci13** monitoring system. The simulator comprises 2026 lines of code spread across 20 source code files. To keep AUTOBAHN 1.0 runtime within reasonable ranges, we used the first 1M of the `batik` trace file from the DaCapo benchmarks **Blackburn06** as the representative input. For this case study, we lowered the *absenceImpact* threshold to 1% because `gcSimulator` does not have many hot spots at higher thresholds. Once AUTOBAHN 2.0 was done optimizing, we evaluated the resulting program on larger trace file sizes of 100M and 500M. Because running the

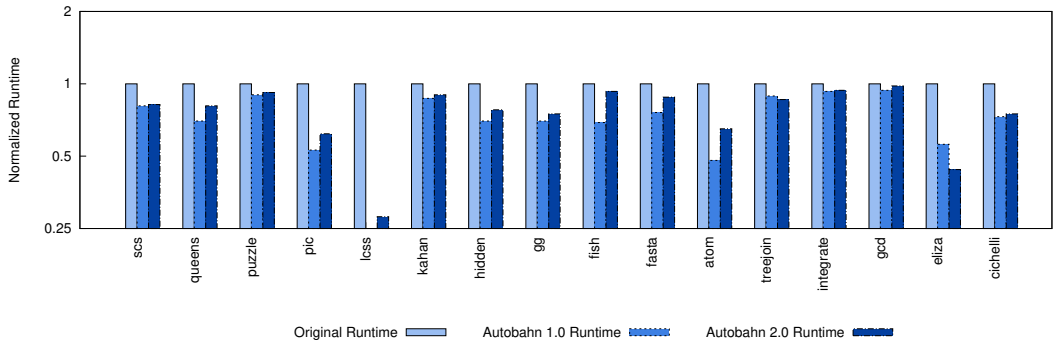

FIGURE 5.15: Runtime performance ratios of AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the first 16 of 32 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using memory allocations as the profiling metric. The x-axis is on a base-2 log scale.

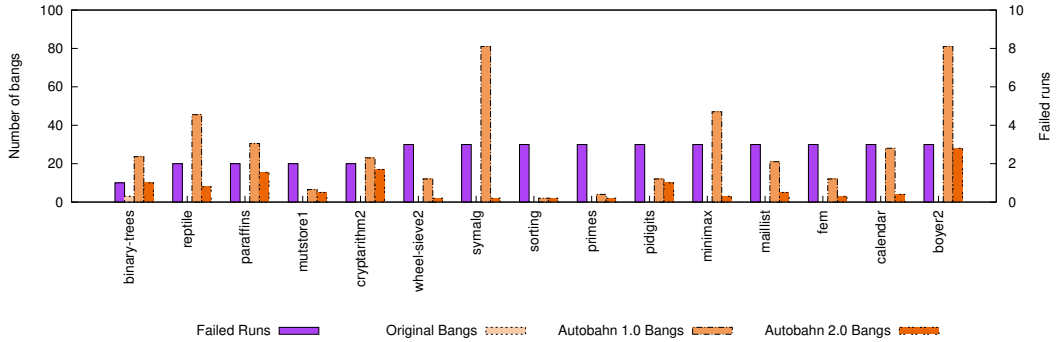

FIGURE 5.16: Number of bangs generated by AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the remaining 15 of 32 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using memory allocations as the profiling metric. Failure rate out of 10 runs is shown. Benchmarks are sorted in increasing order of failure rate. Columns that exceed the maximum axis value are labeled with their actual values.

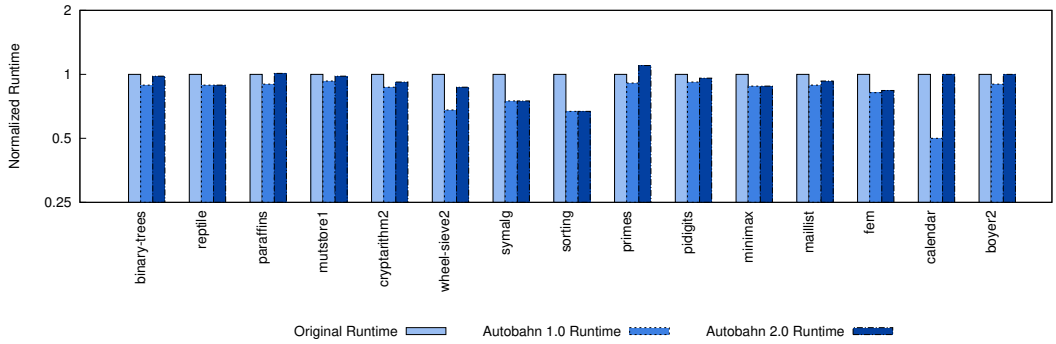

FIGURE 5.17: Runtime performance ratios of AUTOBAHN 1.0 vs. AUTOBAHN 2.0 across the remaining 15 of 32 benchmarks that AUTOBAHN 2.0 successfully optimized at least once using memory allocations as the profiling metric. The x-axis is on a base-2 log scale.

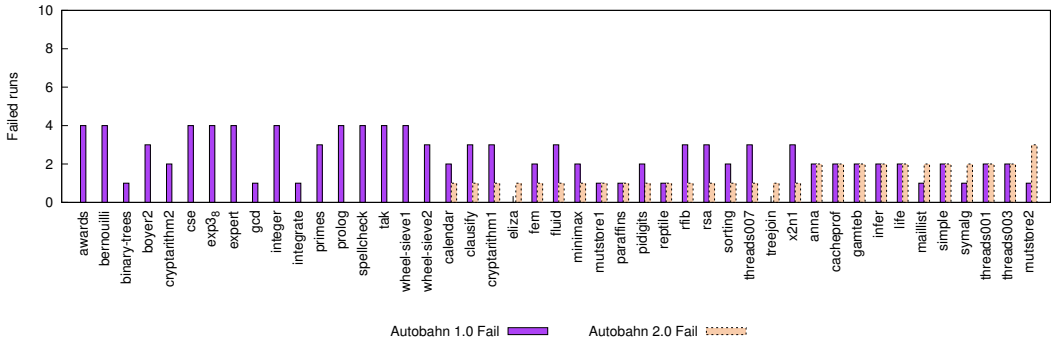

FIGURE 5.18: Frequency of failure attributable to AUTOBAHN 1.0 versus AUTOBAHN 2.0 across the 32 benchmarks that AUTOBAHN 2.0 sometimes successfully optimized using memory allocations as the profiling metric. Benchmarks are sorted in increasing order of AUTOBAHN 2.0 failure rate.

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
	500	68.98	690
AUTOBAHN 2.0	1	0.23	125
	100	15.75	125
	500	81.66	125

TABLE 5.2: Performance results for `gcSimulator`.

simulator on the full 6184M trace size took too long with or without optimization, we did not record results for the full trace.

AUTOBAHN 1.0 produces results that run faster on representative input as well as on larger trace files. AUTOBAHN 2.0 was able to generate similar performance results with many fewer bangs (125 vs. 690), as demonstrated in Figure 5.2. While Figure 5.2 presents results from running AUTOBAHN 2.0 using runtime as the profiling metric, we also ran the same experiment using heap allocations as the profiling metric. We did not find a significant difference in using either profiling metric.

Chapter 6

Related Work and Future Work

6.1 AUTOBAHN 1.0 and Other Methods of Managing Laziness

The current strictness analyzer in GHC uses backward abstract interpretation to identify locations that can be eagerly evaluated **Sergey14** The analysis is sound but necessarily approximate because it is static and the underlying property is undecidable. The analysis only marks locations as strict if it can guarantee termination on all possible inputs, not just realistic ones, since it is part of the compiler. Like AUTOBAHN 1.0, AUTOBAHN 2.0 leverages the advantages of being dynamic and not attempting to guarantee termination on all inputs. Instead, AUTOBAHN 1.0 allows users to decide if the annotations are safe on the intended inputs.

Other approaches for reducing laziness include Strict Haskell **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 **Chang14** take the opposite approach: they start with a program written in a strict language and inserts laziness annotations 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 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. We have occasionally seen programs in our experiments that may be more accurately profiled by heap usage instead of time and allocation profiles. Although it may be difficult to predict which programs' hot spots are more prominently associated with heap usage, AUTOBAHN 2.0 could run both profiling systems for a program and compare the profiles before selecting the more suitable one to use. Another potential improvement is implementing thresholds that automatically adjust themselves based on profiling results. Our experiments show that the ideal values for *hotSpotThreshold* and *absenceImpact* thresholds vary by program to program. Adopting more flexible thresholds that automatically adjust themselves after inspecting the profile in the pre-search phase might yield better results than using set values or asking users to provide them.

Chapter 7

Conclusion

7.1 Conclusion

Laziness is a double-edged sword: While it provides many benefits, excessive laziness can cause poor performance. Strictness annotations allow programmers to force eager evaluation, but its use is limited to experienced programmers with high levels of expertise. AUTOBAHN 1.0 uses a genetic algorithm to automatically infer annotations to improve 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 focus the search space and eliminates useless bangs in a post-processing phase. Experiments show that AUTOBAHN 2.0 removes 90.2% of bangs that AUTOBAHN 1.0 recommended with only a 15.7% optimization degradation on the NoFib benchmark, and 81.8% of the bangs with the same 15.7% optimization degradation on the `gcSimulator` case study.