Revealing Compiler Heuristics through Automated Discovery and Optimization

Revealing Compiler Heuristics through Automated Discovery and Optimization

Volker Seeker†, Chris Cummins†, Murray Cole*, Björn Franke*, Kim Hazelwood†, Hugh Leather†
† *Meta AI* - vseeker@meta.com, * *The University of Edinburgh*

***Abstract*—Tuning compiler heuristics and parameters is well known to improve optimization outcomes dramatically. Prior works have tuned command line flags and a few expert identified heuristics. However, there are an unknown number of heuristics buried, unmarked and unexposed inside the compiler as a consequence of decades of development without auto-tuning being foremost in the minds of developers. Many may not even have been considered heuristics by the developers who wrote them. The result is that auto-tuning search and machine learning can optimize only a tiny fraction of what could be possible if all heuristics were available to tune. Manually discovering all of these heuristics hidden among millions of lines of code and exposing them to auto-tuning tools is a Herculean task that is simply not practical. What is needed is a method of automatically finding these heuristics to extract every last drop of potential optimization.**

In this work, we propose Heureka, a framework that automatically identifies potential heuristics in the compiler that are highly profitable optimization targets and then automatically finds available tuning parameters for those heuristics with minimal human involvement. Our work is based on the following key insight: When modifying the output of a heuristic within an acceptable value range, the calling code using that output will still function correctly and produce semantically correct results. Building on that, we automatically manipulate the output of potential heuristic code in the compiler and decide using a Differential Testing approach if we found a heuristic or not. During output manipulation, we also explore acceptable value ranges of the targeted code. Heuristics identified in this way can then be tuned to optimize an objective function.

We used Heureka to search for heuristics among eight thousand functions from the LLVM optimization passes, which is about 2% of all available functions. We then use identified heuristics to tune the compilation of 38 applications from the NAS and Polybench benchmark suites. Compared to an `-Oz` baseline we reduce binary sizes by up to 11.6% considering single heuristics only and up to 19.5% when stacking the effects of multiple identified tuning targets and applying a random search with minimal search effort. Generalizing from existing analysis results, Heureka needs, on average, a little under an hour on a single machine to identify relevant heuristic targets for a previously unseen application.

***Index Terms*—Search Methodologies, Compiler Optimization, Differential Testing**

I. Introduction

Modern compilers offer many configuration parameters to fine-tune the optimization of targeted application binaries. Default configuration settings, however, do not generalize well [1], and finding the best configuration for specific application targets is an open research problem. Iterative compilation [2]–[4] attempts to automatically tune compiler pass parameters and orderings to achieve high execution speed and small binary sizes when targeting individual applications. The results gained with this method can be tremendous and are usually significantly better than using default settings. However, iterative compilation and other auto-tuning approaches rely on configuration options already exposed by the compiler developer, for example, via command line parameters. Alternatively, they need developers to identify and parameterize relevant heuristics in the source code by hand.

Most research on hand-picked and auto-tuned compiler heuristics focuses on a few examples such as inlining [5]–[7], loop unrolling [8] or vectorization [9]. There are likely many more heuristics used in other parts of the compiler that have been added over the years and have rarely been changed or evaluated since. Most of them are unmarked and difficult to identify, while some were likely not even intentionally designed to be heuristics. As a result, many optimization heuristics remain inaccessible to modern auto-tuning tools and, with that, many opportunities for potential performance gains.

The LLVM compiler framework [10] has millions of lines of code. Combing through all of them manually to identify relevant heuristics is a very impractical task. It would require significant manual labor while working with a moving target as compiler developers add new functionality daily. We need a tool that automatically finds all existing heuristics in the latest compiler version and determines their potential for optimizing a targeted objective. Such a tool would be of tremendous value for application developers to optimize individual programs and compiler developers to find high-value targets for optimizing the compiler.

In this work, we propose Heureka, a system that automatically finds functions in the LLVM compiler source code that encapsulate heuristics. Specifically, we target *LLVM Optimization Passes* (OPT), which are at the core of optimizing code during compilation with the LLVM toolchain. Heureka instruments targeted compiler functions with wrapper code that allows output values of function calls to be modified. Using Differential Testing, Heureka then automatically probes different output values for the instrumented function during compilation and determines which value ranges the calling code accepts as valid function output. If our system can find a non-trivial value range for a function in which selected values generate semantically correct application code and improve an optimization objective, the function is considered a heuristic. Otherwise, Heureka discards the function and evaluates the next one. Our heuristic search looks for functions relevant to reducing the size of generated binaries. The binary

size objective, however, only serves to illustrate our method. Heureka is designed to allow different optimization objectives as well, like application runtime.

We leverage binary size heuristics suggested by Heureka to focus on two use cases:

Application Optimization We use Heureka to suggest heuristics and corresponding value ranges relevant to reducing the binary size of individual applications. Focusing on only one application at a time, Heureka can automatically suggest heuristics from thousands of candidate compiler functions that can heavily tune the targeted application, even if they might not be safe for others. We then use a random search to tune the application size by selecting output values for suggested heuristics during compilation.

Compiler Optimization Based on suggestion and tuning results for our evaluated applications, we can identify and rank compiler functions according to their potential for binary size reduction. Furthermore, we can provide identified heuristics to compiler developers who can verify their correctness, add potentially required safety guarantees, and use our existing instrumentation and provided value ranges to plug in their own tuning tools with minimal effort.

a) Results: With our technique, we analyzed eight thousand functions from OPT, which is about 2% of all functions available for our purposes in LLVM. We were able to reduce the binary size of 38 benchmarks from the NAS and Polybench/C benchmark suites [11], [12] by an average of 7.7% compared to an `-Oz` baseline by tuning only individual heuristics. Combining the effects of multiple heuristic suggestions using a random search achieved additional savings as high as 19.5%. The heuristic search for previously unseen applications was executed automatically and needed on average a little under an hour if executed on a single machine using pre-existing analysis results we collected for other applications. This evaluation time is in line with other auto-tuning approaches [13]–[16]. In addition, we identified and manually validated a set of heuristics that showed high binary size savings for most applications without breaking compilation or generated binaries for any of them during tuning.

We make the following contributions:

- A novel method to automatically identify functions encapsulating heuristics in the LLVM compiler and their accepted output value ranges.
- A framework to expose those functions and automatically search their output space to optimize the binary size of individual application targets.
- An instrumentation compiler pass and extension library to instrument functions in arbitrary programs[1].
- A full dataset1 with the heuristics we identified for targeted applications, categorized as *obvious*, *indirect*, *data field*, and *unsafe*. Examples are discussed in Section VII and Appendix B.

[1]see online repository at *redacted-for-review*

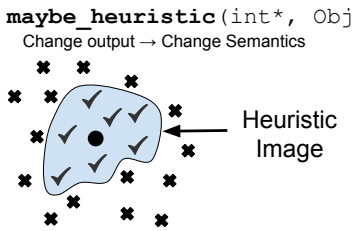


Fig. 1: Potential heuristic function and corresponding image. All marks are return values of the function for various test inputs. The large circle indicates an original unmodified return value, while the crosses and ticks indicate unacceptable and acceptable modified return values for the same original. The space of acceptable outputs across all original values combined is called the *image* of the function.

II. Heureka Overview and Terminology

In this section, we give an overview of how Heureka identifies heuristic functions and introduce the required terminology. On a high level, Heureka walks through individual compiler functions and uses a Differential Testing approach to explore which value modifications are possible for function outputs without breaking the compilation of a target application or its generated code whilst also showing objective improvements such as a smaller application binary. Functions satisfying those two properties count towards our list of identified heuristics.

a) Function Image: A central concept of Heureka's approach is the *function image*. The image of a function is the range of output values accepted by the function's calling code (see Figure 1). Any of the values within this range can be chosen as output without changing the generated program's semantics, i.e. without leading to compiler or program crashes or invalid program results. For example, if a function decides how often to unroll a loop and returns this decision as an integer value, then returning negative integers may invalidate the calling code or generate invalid binaries – in this case, the function image would be all non-negative integers. The possible output a function can return following its current implementation does not always fully overlap with the function image, i.e. the calling code might accept a wider range.

b) Encapsulated Heuristic: Heureka aims to automatically identify functions for which we can approximate their corresponding function images and find at least one new output value within the function's image that improves a chosen optimization objective. We call such a function an *encapsulated heuristic function* – encapsulated because the heuristic is embodied in a function, rather than some more arbitrary region in the code. Identified heuristic functions and their approximated function images are then exposed to auto-tuning tools.

c) Optimization Objective: The *optimization objective* is the metric we aim to improve with identified heuristics. We can, for example, tune heuristics for program execution time, memory footprint or generated binary size. In this study,

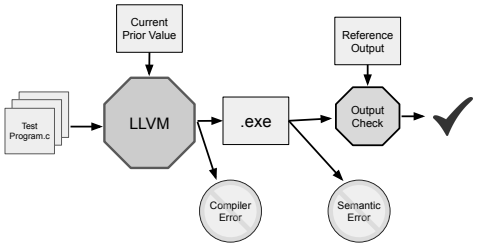


Fig. 2: Differential Testing setup to identify a fitting image template, aka. Prior, for a targeted LLVM function. A range of test application code is compiled using a modified compiler. During compilation, specific output values are forced for instrumented functions as selected by the Prior currently being evaluated. A forced output value is considered acceptable for its corresponding function image, if the test code can be compiled successfully, the generated test binaries execute without fail and the expected test output is produced.

we chose generated binary size compared to an `-Oz` baseline because it is robust, straightforward to record.

d) Output Channels: A function can have more than one way to return output. We call these *output channels*. Each channel can have its own function image. On a high level, an output channel can be a function's return value or a reference or pointer parameter[2]. In Figure 1 the function `maybe_heuristic` has three high level output channels, namely the integer return value and the integer and Object pointer parameters.

e) Priors: For most functions, it is impossible to know what their image looks like from the source code alone. It is also infeasible to test all possible output values for all possible inputs to determine which values belong to the image and which ones do not. Instead, we devise image templates we call *Priors* which express what an image could look like in a testable way. For example, a Prior may be that all integers within a minimum and maximum bound are part of the image. We use effective search strategies to test only the values we need to confirm or discard a Prior and its parameters for a function output channel. We can increase the confidence we have in the validity of our accepted Priors by increasing the number of test cases we evaluate individual output values against. In the context of a compiler, that would mean, the more application code we compile and verify successfully against compiler function modifications, the more confident we are in our accepted Priors.

f) Probes and Test Cases: To test if a value is part of a function's image, we use a Differential Testing approach [17] we call *probing*. Differential Testing enables automatic evaluation of a test outcome by comparing it to the outcome an unmodified reference version of the tested software produced. If they are the same, the test passed. Figure 2 shows how we evaluate a probe using Differential Testing. A single probe is tied to a specific output modification and Prior. The Prior currently being evaluated determines the next output modification to be probed for a targeted function according to its search strategy. We dynamically instrument the target function by wrapping it with code that allows Heureka to control the function's output (see Appendix A). We force the target function to return a modified value as output for one of its output channels. This way, we emulate a different compiler version to compare against where the benefits of Differential Testing come into play. A range of *test application code* is compiled to exercise the modified function with the forced return value. The compilation then either succeeds and generates a program binary or fails and crashes. If a binary was successfully produced, we execute it and evaluate its output against a reference to determine if it is semantically correct or not. If a valid program binary was produced, the probed value is considered acceptable as part of the current Prior's image and we can try the next one. If the generated binary is invalid or compilation fails, the Prior or its current parameterization will be discarded. To cap runtime and avoid endless loops, we employ a configurable timeout of sixty seconds as the upper bound for the modified compilation and runtime of application code.

g) Tuning Targets: It is possible that a function is a heuristic but already optimally tuned. Therefore, in addition to validating the current output for a probe, we also record our chosen objective metric. If we can record an improvement and find a valid Prior, we note the corresponding function and output channel as a suggested *tuning target*; if not, we go to the next function. Using the described probing process, we go through a range of selected compiler functions and their output channels and try to fit one or more of our available Priors.

h) Correctness: Heureka does not make correctness guarantees for the generated binaries. Instead, it relies on tests and verification provided by the application developers. More tests reduce the probability of suggesting heuristics that break generated binaries but do not entirely prevent them. However, binary size is of such critical importance to many in industry [18]–[21] that application and compiler teams, which dedicate many person years to reducing code size, have exhausted most of the techniques in the literature and still need to deliver wins. For those teams, any promising leads, even those which may require further investigation by experts, are extremely welcome. Our technique can help application teams to find optimizations that, while unsafe in general, are safe for their specific application. For compiler developers, it can find new opportunities that would be too difficult to find without automation. We suggest enough correct heuristics to be worth the effort and additional automated testing can reduce invalid suggestions still further. The approach of suggesting potentially unsafe optimizations has been used to great effect in prior literature [22], [23].

III. Priors

To find tuning targets in a program, we approximate images for all output channels of a function using Priors. In this work, individual output channels of a function are independent and

[2]There are other ways of producing heuristic output. For example via global variables or by IO. We do not currently consider these.

TABLE I: This table shows Priors used in this study to approximate output channel images for target functions and the value ranges they describe. r indicates the original output value while *min*, *max*, θ, ϕ, α and β are Prior parameters that are fitted as accurately as possible during Prior evaluation.

Prior	Range
Boolean	$\{0, 1\}$
All Integers	$[MIN_{int}, MAX_{int}]$
All Reals	$[MIN_{real}, MAX_{real}]$
Integer / Real Range	$[min, max]$
Integer / Real Offset	$[r - \theta, r + \phi]$
Integer / Real Scale	$[r(1 - \alpha), r(1 + \beta)]$

can have different images. Output channels of a function are described in detail in Section IV. For now, all we need to know is that an output channel is bound to a primitive type such as a 32-bit integer or a real value. This type, depending on the architecture, describes the maximum possible value range that can be part of the image. The actual image might be significantly smaller down to a single allowed value – the original value used by the function. Essentially, a Prior describes the shape of an output image and can be parameterized to approximate the actual value range of the image, e.g. the image encompasses values between zero as a lower bound and ten as an upper bound.

Priors can describe simple images such as "all 32-bit integer values are allowed" or more complex ones such as a bounded or scaled range around the function's original output value. Table I lists all Priors we use in this study. They are based on our knowledge of how typical heuristic functions behave, but depending on the target compiler, additional Priors can easily be added. We support Boolean, 8, 16, 32 and 64-bit Integer and 32 and 64-bit Real value Priors.

Typically there will be far too many candidate Priors and Prior parameterizations to search exhaustively. We, therefore, need efficient search strategies to determine if a Prior approximates a channel's image correctly or not. The following describes our approach to testing and searching for Priors, including sharing of information between searches, and effective ordering of searches.

Priors are selected depending on the primitive type of the output channel. The order of Prior evaluation is depicted in Figure 3. To determine good starting points for our Prior evaluation, we initially compile each test program without any output modifications to the targeted function ①. This way, we can record original output values. A targeted function can be called multiple times during compilation of a single program which means after this initial first step, we usually have a large sample of original values available. Before we start evaluation we also record a baseline optimization objective for test programs against which we compare modified values to determine possible improvements.

Our Priors are divided into two groups: "absolute Priors" where the range is agnostic of the original output and simply probes absolute values (Boolean, All and Range) and "relative Priors" which consider the original output and probe values relative to it (Offset and Scale). To avoid unnecessary work, we use the initially recorded original output to decide if we should consider absolute Priors or directly start with relative ones ②. We do this by randomly selecting a set of recorded originals including the minimum and maximum we observed. We probe this set against our test application code by forcing the currently probed value as output for all executions of the targeted compiler function. This way, we use a value that worked for some function executions and check if it works for all of them. If this test is successful, we can work with absolute values and continue with the All Prior ③ otherwise, we go directly to the relative Offset and Scale Priors ⑤.

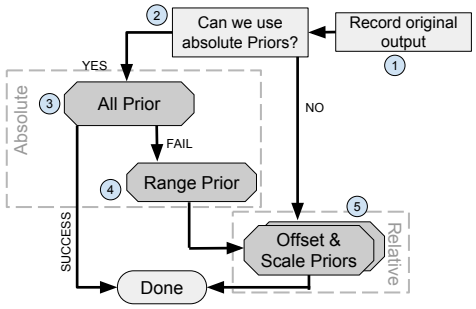


Fig. 3: This Figure shows how we search for fitting Priors depending on the type of the targeted output channel.

The Boolean Prior has a very small value range which we can test in its entirety. We simply probe both possible values and considered it a fitting approximation of a boolean output channel image if both values are valid. To simplify the flow chart in Figure 3, we did not include the Boolean Prior.

For all types other than boolean, we start with the corresponding All Prior ③. It is a superset of all other Priors which is why we evaluate it first if absolute values are possible. It assumes that all integer or real values (depending on output channel type) are part of the image and we probe a set of sample values to validate this assumption. Among the values we test are the numerical bounds of the corresponding type, such as MAX_{int} and MIN_{int} for an integer type, as well as interesting values like zero, plus and minus one. Additionally, we sample a random selection of test values from the entire available value range. The number of samples can be configured and we are using a value of thirty in this study. We make certain that we keep a history of probed values in between Prior evaluations so that we don't probe values more than once. If all of the All Prior's sample values are evaluated without failing test cases, it is assumed to approximate the output channel's image correctly.

Should it fail, we continue with the Range Prior ④ which assumes that all integers / reals within a given interval between *min* and *max* parameters are part of the output channel's image. We initialize the range search from previously observed samples we probed so far. To do that, we take the lowest and highest successfully probed values that do not have any failing observations in between. We then extend the initial bounds with a binary search strategy first towards positive values and

```
bound_min=0
initial=1
bound_max=INT_MAX
while (validation is not successful):
    final_bound = extend_bound_until_fail_or_max(
                      bound_min, initial, bound_max
    )
    targets = generate_random_targets(
                      NUM, bound_min, final_bound)
    validate_current_bound(targets)
    if validation fails:
        bound_max = targets[first failed test]
        initial = targets[last valid test]
return final_bound
```

Fig. 4: Binary search used to fit a bound parameter for a Prior.

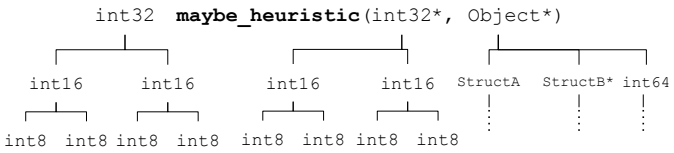


Fig. 5: This figure illustrates low level output channels for a sample function. To save space, LLVM can combine two or more individual parameters or fields of a struct in the actual source code into a single one in LLVM-IR. Therefore we need to assume, that a single output channel can in fact be a combination of multiple ones. Every node in the depicted trees indicates a potential output value which is evaluated by our methodology.

then towards negative values until we narrow the bounds down to the last accepted value in the interval or the maximum possible value in the available numerical range for the output channel's type. To increase our certainty in the uncovered interval, we probe interval bounds whenever our binary search determines a final bound as well as a set of randomly selected values within the new interval. Should the validation fail, we re-initialize the binary search according to validation results and execute the search again with refined parameters. If the interval contains a single value only, we do not consider the Prior to be successful as it gives us no additional information to tune an output value beyond the original one.

Lastly, we evaluate the relative Offset and Scale Priors. They work with the original value rather than being oblivious to it. For these Priors values forced as outputs for target functions are relative to the original output of the target function execution. If the channel would originally return value r, the Offset Prior's search strategy would try to find a lower bound θ and an upper bound ϕ around r which could be added to the original value and still produce an acceptable result ($[r-\theta, r+\phi]$). It starts with a lower and upper bound of zero and extends first the upper bound as high as possible using a similar binary search as the Range Prior and then the lower bound. It stops as soon as it finds the last working offset that can be added to original output values. If both lower and upper bound offsets are zero, the Prior is not considered to be successful. The Scale Prior's strategy scales the original value according to an α and a β parameter. As before, it uses a binary search strategy to push the α and β factors as far as possible ($[r(1-\alpha), r(1+\beta)]$). Again, if both parameters are zero, the Scale Prior is considered unsuccessful.

Figure 4 shows the binary search algorithm used to narrow down a single bound of Range, Offset or Scale Prior in pseudo code. If exemplified for the positive offset of an Integer Offset Prior, `bound_min` is initialized to a minimum allowed bound which would be zero while `initial` is initialized to some initial starting value like one. `bound_max` is set to the numeric maximum of the corresponding channel's data type. In our example, this would be a maximum positive integer value. Now, a binary search finds the first failing probe going forward from `bound_min` towards `bound_max` and returns it as `final_bound`. If all probes are successful along the search, `bound_max` is returned as final bound. To increase our certainty in the selected positive offset, we probe a sequence of `NUM` validation targets in between `bound_min` and the current `final_bound`. The number of targets can be configured and we used a value of thirty for our study. If any of the validation probes fail, the binary search is re-initialized and started again. The new `bound_max` is then set to the failing probe value. The last succeeding value right before the new `bound_max` is used as new `initial`. Once validation is successful, a final bound is returned. In practice, this extra probing works well for us to increase confidence in the ultimately determined interval for the Prior.

Not all Priors are mutually exclusive and we may find none, a single or multiple Priors that are accepted as approximation for a function image. If no Priors are identified for any function output channel, the function is assumed not to be an encapsulated heuristic. The same is true if no significant changes in the objective can be observed while determining a fitting Prior. If we can fit a Prior but do not observe objective improvements, we still suggest the function and its identified Priors. In a post processing step, this information can be used to search the approximated image more thoroughly which can still reveal an improved objective. Also, the results indicate that modifications within the identified Prior parameters are safe when used for compiling tested programs even though they don't yield objective rewards.

IV. Output Channels

To evaluate a target compiler function, we initially determine what the possible output channels of the function could be. A function's output channels are the return value or parameters that can be used to pass data back to the calling code such as pointer and reference parameters. In Figure 5 the function `maybe_heuristic` has a total of three top level output channels: the integer return value as channel A, the integer pointer parameter as channel B and an `Object` struct pointer as channel C. This does not mean that the function actually uses all of them for output.

Next to the obvious top level output channels, we consider more complex cases where top level channels break down

further into lower level channels that we evaluate in addition. The breakdown of channels for the example function is sketched in tree form in Figure 5. Imagine a function calculating loop unroll cost which returns a struct containing two integer values. One indicates the cost of unrolling a loop while the other indicates the cost of running the loop without unrolling. Based on this cost, the loop unroll factor is calculated by the calling code. So modifying either of those two values can affect a binary size objective. Therefore, in the case of a struct output, all of its member fields can be an output of a potential heuristic and need to be evaluated. You can see this exemplified for the `Object` struct pointer in the sample function. Structs in turn can contain other structs and primitive types where the output paths can branch further.

To modify function outputs dynamically, we use an LLVM compiler pass to add lightweight static function instrumentation to the targeted compiler (see Appendix A). That means we use an unmodified vanilla LLVM compiler (VLLVM) to build our LLVM compiler target (MLLVM) we then modify and evaluate with Heureka's heuristic search. The Instrumenter pass modifies the LLVM Intermediate Representation (LLVM-IR) of MLLVM's source code by placing wrapper code around targeted functions. On IR level, VLLVM applies optimizations where multiple primitive types in the original high level language code can be combined into a single type in LLVM-IR. Therefore, if Heureka sees an integer value with 32 bits as output, it needs to assume that it can be a combination of two 16-bit integer values or a 16-bit value and two 8-bit values, etc. Heureka analyses possible combinations individually and provides fitted Priors for all, if successful. Combinations of types can be broken down all the way to bit level. If a heuristic manipulates bit fields, a prior considering this level of granularity can be beneficial. We left it at the byte level as an informal code review led us to believe bit fields are rare in the LLVM code base.

If considering Figure 5, every node in the depicted trees that contains a primitive type is an output channel. Therefore, the number of output channels Heureka considers for a targeted function can be large and complex, especially, if struct types are involved. Heureka is flexible in the way output channels are constructed and additional strategies can be added by developers if required.

Considering, that we are looking for potential heuristic functions, not all reachable channels in a tree as described in Figure 5 would be sensible for returning heuristic results. Hence, Heureka prunes output channel trees to improve search efficiency. It is, for example, unlikely that compiler heuristic results would be returned deeply nested into a struct output channel. Therefore, we apply a maximum depth to our channel selection algorithm and allow at most a single dereference along a path on an output channel tree before we stop looking. Additionally, we have further backstops in place to rule out unlikely output channels such as function pointers or the content of arrays. We also skip over function parameters that are marked as `const` in the original C++ code since those cannot be modified by the function as dictated by the language syntax. Furthermore, while recording original outputs for a channel in the initial Prior search step (see Section III), we rule out function reference or pointer parameters as output channels that are never modified by observed target function executions.

V. EVALUATION

In this section, we describe the setup we use to tune LLVM heuristics with Heureka for individual application targets and present our findings. Additionally, we discuss specific functions we identified as heuristics in the LLVM compiler that showed high binary size savings and worked well for most of our targeted applications.

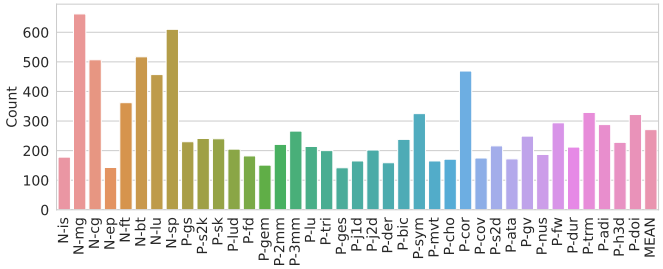
A. Experimental Setup

In our evaluation, we aim to identify and tune heuristic functions among the optimization passes of the LLVM Clang compiler version 10.0.1. Specifically, we target the first eight thousand functions from the Analysis and Transformation passes [24] that can be found as code modules in the LLVM source code directories `llvm/lib/Transforms/` and `llvm/lib/Analysis/`. This number is equivalent to roughly 2% of all available functions in the LLVM Framework we considered for our purposes. We ignore functions that are not interesting to our study such as standard library functions, C functions, destructors, main functions or functions with unknown types. Our instrumentation uses a custom type system which is based on LLVM's internal type system. We support most common types that are also supported by LLVM but not all of them yet, e.g. vector types.

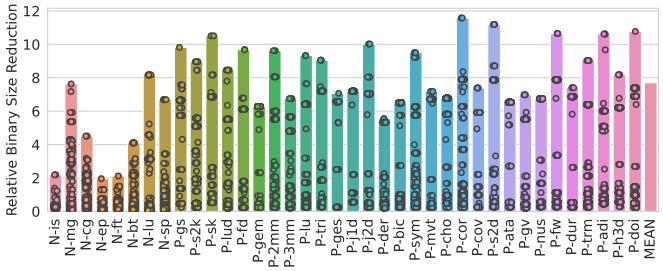
As an optimization objective, we chose the size of an application binary generated by the Clang compiler, where smaller means better. Specifically, we measure binary size using the *llvm-size* tool [25] adding up its first two output columns `text` and `data` when using the *Berkley* output format. As our baseline, we consider the binary sizes generated by the unmodified vanilla Clang compiler when setting it to its most aggressive binary size reduction level with the `-Oz` flag.

Heureka requires tests to verify semantic correctness of optimized applications, hence we chose the NAS and Polybench/C benchmark suites [11], [12] which come with reference output and correctness tests provided. We augmented available tests with additional checks of output values and used all 38 available applications for our experiments. To evaluate individual probes when fitting Priors, we use an instrumented compiler (MLLVM). We build object code from benchmark source files with a direct `clang -Oz -c` call. We then link object files to a final binary using an unmodified vanilla compiler (VLLVM). That means, while building object files with Clang we modify the output of instrumented functions in the applied code optimization passes that are active for the `-Oz` flag.

The validation against a reference output for NAS benchmarks is integrated into the benchmark source code. To avoid modifying and thereby influencing the validation itself when compiling a benchmark with MLLVM, we decoupled validation



6.1: Distribution of tuning targets identified per benchmark application. Bars can include targets that are shared by one or more benchmarks.



6.2: Maximum binary size reduction found during heuristic search relative to `-Oz` baseline when considering single heuristics only. Overlaid circles indicate all of the tuning targets we could identify for this benchmark at their maximum observed binary size reduction.

code from benchmark code. The code to validate benchmark results is kept in separate source files which are compiled independently into object files using VLLVM. They are then linked to a final benchmark binary together with the object files which were compiled with MLLVM. For Polybench benchmarks, we activate full result output to `stdout` with `POLYBENCH_DUMP_ARRAYS` and compare it to a pre-recorded reference. For all benchmarks, we use the smallest data sets during execution since they are sufficient to measure binary size and allow fast evaluations.

To further improve the validity and robustness of our results, we minimize all benchmarks before we optimize their binary size with Heureka. We use C-Vise which is a super-parallel Python port of the C-Reduce source code minimizer [26], [27]. To guide minimization, we use an *interestingness* score that follows the same benchmark validation and input data sets we consider when fitting Priors during Heureka's heuristic search. In addition, we make sure that resource allocation of file descriptors and memory is preserved using wrappers around relevant STL functions such as `malloc`, `free`, `open`, etc. Minimization increases the probability that our modifications during compilation affect code that lies on the execution path of the targeted application and that we are not reporting results where code was merely dropped from irrelevant sections. Also, it is more likely that faulty code modifications will be caught during program execution and validation. NAS benchmark binaries are with an average of 22 KB baselines after code minimization generally larger than the Polybench benchmarks with a 3 KB average.

Individual output channels for instrumented functions can be evaluated independently of each other which allows us to heavily parallelize and distribute this process to multiple nodes of a compute cluster. The cluster used in this study consists of approximately 8000 cores with up to 3 TB of memory per node. The cluster uses the Univa Gridengine batch system [28] on Scientific Linux 7. We were assigned a share of 400 cores and 1.6 TB RAM for our study.

B. Application Optimization

Our first use case is to optimize individual applications by identifying compiler tuning targets that achieve binary size savings by themselves and in combination using a random search approach. We present tuning results using two evaluation approaches for each application. The first approach is to go through all eight thousand target functions which include a total of 150 thousand target channels and required over one million probes to be executed to find fitting Priors. We call this the *Full Evaluation Set* and run the Heureka heuristic search over this set for each of our 38 benchmark applications in turn.

Using the Full Set, we suggested an average of 271 tuning targets per benchmark (see Figure 6.1). Most NAS benchmarks (prefixed with **N**) have higher tuning target counts because they have larger code bases than the smaller Polybench applications (prefixed with **P**) and thus more optimization opportunities. Figure 6.2 shows a breakdown of the maximum binary size savings we could find for all benchmarks during the heuristic search with an average of 7.7%. The height of the bar indicates the maximum improvement we could observe while the circles show individual tuning targets for that benchmark at their maximum observed binary size reduction. While most binary size reductions are only a fraction of a percent, we can find some that go as high as 11.6%. We discuss specific examples of high reward tuning targets in Section V-D.

The heuristic search does not try to find optimal output values but only requires at least one probed value to improve the objective in order to identify a heuristic (see Section II). Its goal is to suggest heuristics and corresponding value ranges for subsequent auto-tuning of the targeted application. Hence, our random search tuning using suggested heuristics achieves even higher savings (see Section V-E).

The Full Set only focuses on single applications and therefore validates compiler modifications only for their application target using the specified input set and validation methods we described above including prior code minimization. Application developers using this technique for their code can extend the validation to increase their confidence in the validation results. As an example, let's consider P-cor which shows the highest saving results of 11.6%. We looked at the binary that was generated at the corresponding tuning point suggestion in more detail to confirm Heureka's validation and to apply further analysis. All tests pass and the generated output is as expected but `valgrind` points out memory leaks that do not exist in the original application and were introduced by the tuning process. The second highest successful tuning target for this benchmark does not show memory leaks and

achieves 8.4%. In such cases, application developers can discard unwanted heuristic suggestions (see Section V-D for a more detailed discussion).

In our second evaluation approach, we aim to transfer tuning target findings from existing results to previously unseen applications. This way we can improve evaluation performance and identify tuning targets and corresponding Priors that generalize well to other applications. To do this, we combine all successful tuning targets and Priors identified using the *Full Evaluation Set* for individual applications to two new evaluation sets, namely the *Reduced Evaluation Set* and the *Restricted Evaluation Set*.

For the Reduced Set, we take a conservative approach and generate the union of all tuning targets and Priors that showed objective improvements. For the Restricted Set we only allow safe tuning targets, i.e. targets that are identified for all of the other applications, share the same Priors and showed improvements for at least one of them. If a tuning target was not used during compilation by some applications but show required properties for others, we also consider it safe. As described in Section IV, Heureka splits the data type of an output channel into smaller byte ranges in case values have been combined during compilation. Each of those byte ranges is considered a channel. For the Restricted Set we filter out unnecessary type splits. If objective improvements of split channels are exactly the same, we keep only the widest byte range. Additionally, we further prune output channels for targeted functions that we consider to cover the same value in the actual code. For example, when a function returns a 32-bit integer, we would break the channels down into specific subregions of the byte range in case LLVM has combined types (see Section IV). However, when we observe the same objective improvements for all channels, we would only keep the widest byte range.

We generate the Reduced and Restricted sets using cross-validation by holding out an application at a time and combining tuning target results for all of the others. We then look at the results for a heuristic search considering only the targets in the combination sets for the held-out application rather than the Full Set with 150 thousand potential tuning targets. For a held-out application, not all tuning targets in the combination sets will be executed during compilation or manage to achieve rewards. Also, we might miss out on some tuning targets that only work for the new application and not for the ones we have seen so far. However, the evaluation effort is significantly reduced. On average the number of output channels that we need to evaluate for each application drops to 1.1% for the Reduced Set and 0.2% for the Restricted Set of what would be required for the Full Set. Additionally, it allows us to identify tuning targets that generalize well and can be provided to compiler developers.

Figure 7 shows the maximum objective savings we were able to observe for the Reduced and Restricted set during the heuristic search and compares it to the Full Evaluation Set. The Reduced Set results are the same for every benchmark except N-is, N-CG and N-ep which had tuning targets in the Full Set that showed high savings only for them. Due to hold-out validation, those tuning targets are not considered. The Restricted Set still manages to show good saving results for most benchmarks with only a small drop in its average and even achieves the same as for the Full Set for some. This indicates that some of the tuning targets showing high savings generalize well among all benchmarks.

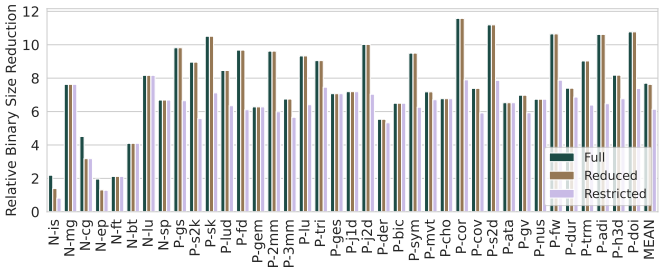
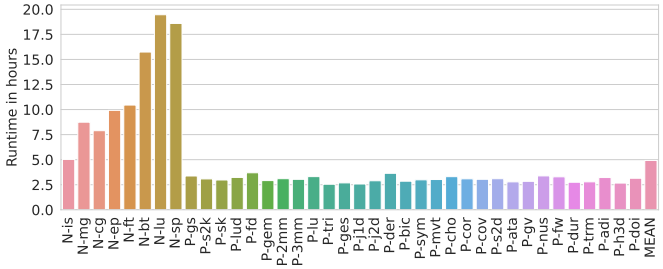


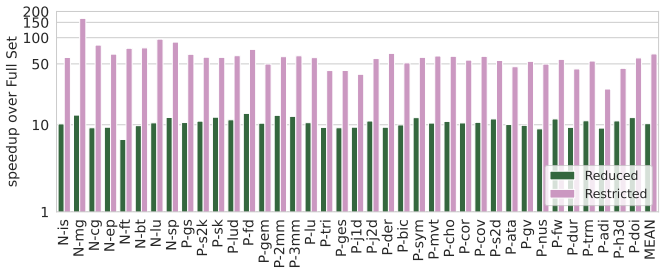
Fig. 7: Relative binary size reduction per application observed during heuristic search when comparing the Full, Reduced and Restricted Evaluation Sets.

C. Execution Time

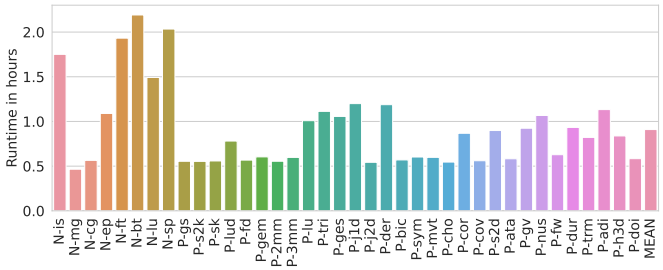
Heureka's execution time depends on which Priors are used, Prior search strategies, output channel pruning and how many functions are considered. Also, the tool itself requires warm-up to set up required resources and has synchronization and bookkeeping overheads in addition to the compile and runtimes of individual probes. Figure 8.1 shows the execution times for individual applications when running them on our cluster. Beyond this, most of the work Heureka does when evaluating output channels is embarrassingly parallel. The average runtime for our evaluation of eight thousand functions with Heureka is five hours. We calculate Heureka runtime by adding up measured compile and run times of individual probes during the heuristic search. For combination set runtimes, we only consider heuristics that remain after subsetting from the Full Set. Figure 8.2 shows the speedup over the Full Set we would see when running our combination sets on the cluster. Those are on average 10x for the Reduced and 65x for the Restricted Set. We ran the heuristic search for the Restricted Set of each application on a single machine with 72 cores and 504 GB of memory (see Figure 8.3) and achieved an average runtime of a little under an hour. Comparing our average runtime for distributed execution of the Restricted set on the cluster to the measurements on the single machine, we observe a speedup of 30x over running the Full Set if normalized to single core performance. Cluster speedups are likely higher due to differences in utilizing existing compute power since we ran individual applications interleaved on the cluster and one after the other on the single machine. When only a few targets were left to explore for a single application, the cluster could execute other applications in parallel while the single machine would wait for them to finish first. Additionally, the random elements in the heuristic search while fitting Priors can affect the outcome duration.



8.1: Heuristic search runtime on 400 core cluster when evaluation the Full Set of functions for each application.



8.2: Speedups of Reduced and Restricted Sets over runtime of the Full Sets on the cluster for each application on a log scale.



8.3: Heuristic search runtime on a single 72 core machine when evaluating the Restricted Sets for each application.

D. Compiler Optimization

Besides the optimization of individual applications, we looked at a second use case for Heureka which is providing heuristics to compiler developers that generalized well for our benchmarks and showed high objective savings. We looked in detail at some of the tuning targets of the Restricted Set over all applications and manually investigated how exactly objective savings were achieved. We generally observe four categories of suggestions for heuristic tuning targets made by Heureka and aim to discuss examples for each in this section and in Appendix B. There are many more and we provide full results for compiler developers to go through in our online repository at *redacted-for-review*.

Obvious Heuristic Obvious heuristics can quickly be identified as such by looking at the source code of the corresponding function. Compiler developers can pick them up and directly use our provided instrumentation and value ranges to apply their own tuning. An example is the function `getInlineCost` in the module `llvm/lib/Analysis/InlineCost.cpp` which calculates a cost for inlining a given function call. Heureka suggests this function and can fit a Prior for the `InlineCost` return value.

Indirect Heuristic An indirect heuristic is a tuning target suggested by Heureka where the corresponding function is not immediately recognizable as a heuristic or was not even designed to be one. However, modifying its output values can positively affect the objective in a subsequent optimization. A good example is the function `bool CallBase::isNoBuiltin` in `SimplifyLibCalls.cpp`. The targeted output channel is the `bool` return value and the corresponding Prior is a simple Boolean Prior. The function generalizes well: All applications execute it, Heureka could fit the Boolean Prior for all of them, and found binary size savings as high as 7.9% for most. According to the source code [10], the function indicates for a `Call Instruction` if the corresponding call is to a built-in library function or not. It is used by the function `optimizeCall` in the class `LibCallSimplifier` which replaces calls to library functions with a more optimal form, for example, it replaces `printf("Hello!")` with `puts("Hello!")`. If the function returns `false`, the replacement of library functions for `printf` with `fwrite` and `fputc` does not happen and the corresponding binary is smaller since fewer function headers are included. We could connect other tuning targets to similar effects such as `sanitizeFunctionName` and `dropLLVMManglingEscape` from `TargetLibraryInfo.cpp`.

Heureka can suggest multiple indirect heuristics that are called in succession to ultimately affect the same optimization. Compiler developers can use these suggestions directly or as indicators to find where tuning can be applied most efficiently. We plan to extend our system to consider such call chains and help compiler developers to make the best choice for their tuning efforts.

Data Field Heuristic We discovered that some tuning targets refer to the `this` argument of an object's member function. Heureka treats the `this` argument as a struct output channel and tries to find Priors for its individual fields. For those suggestions, the tuning target is not the output of a function but a data field in its corresponding object instance. This also means that Heureka is able to track and modify side-effects of functions that affect an object's state. An example is the constructor of `CFGSimplifyPass` in module `llvm/lib/Transforms/Scalar/SimplifyCFGPass.cpp` which is a pass to simplify the control flow graph of a function. Heureka finds that binary size can be reduced if changing the object field `this->Options.BonusInstThreshold`. The `Options` field holds parameters that are used to control transformations performed by the pass. The modified threshold affects the combination of basic blocks and thereby code size.

Unsafe Heuristic Some suggested heuristics are not safe to use and potentially generate incorrect binaries that are not detected by the test set used during the differential testing. An example is the tuning target responsible for the 11.6% binary size reduction of the P-cor benchmark discussed in Section V-B. The corresponding function in the compiler is `bool getLibFunc(StringRef funcName, LibFunc &F)` from the module `TargetLibraryInfo.cpp`. It searches for a library function name and can lead to savings in a similar

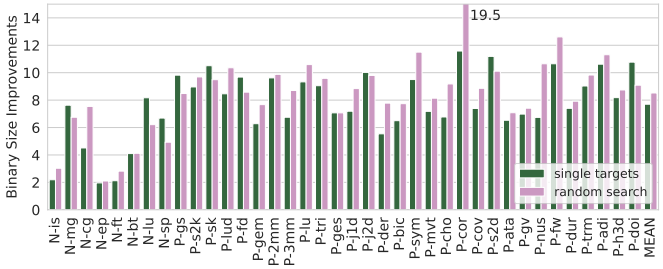


Fig. 9: Relative binary size reduction when combining multiple tuning targets for the Full Evaluation Set using a random search approach.

way as described for the `isNoBuiltin` function. Heureka finds that multiple output channels can be manipulated to achieve binary size savings. Those are the `bool` return value, the size of the `StringRef` argument `funcName`, or via the enum argument `LibFunc`. Modifying the first two is safe and causes no problems in the generated binary. Modifying the enum argument, however, can lead to the removal of `free` calls which causes memory leaks and is therefore an unsafe suggestion. We discovered this when we evaluated suggested tuning targets in more detail by running generated binaries with valgrind and inspecting LLVM bit code.

If the introduced memory leaks are problematic for the application, the tuning target modifying the `LibFunc` argument should not be used. However, if the operating system is sufficient to clean up the memory that was left allocated, the extra binary size savings might be worth using the suggested target after all. To improve the suggestions made by the heuristic search, a valgrind analysis can be added to the differential testing of each probed value.

E. Random Search Tuning

Heureka's heuristic search is only the first step when tuning applications. Here we automatically identify heuristics in the compiler source code and fit Priors to determine acceptable output value ranges (see Section II). During this process, functions are only accepted as tuning targets if we can fit a Prior and they show objective improvements for at least one modified output. This way, we already find improved binary sizes for a tested application as soon as we accept a tuning target. Heureka's value space exploration and heuristic discovery, however, is optimized for efficiently fitting Prior parameters, not for finding the most optimal output modification. Also, it only considers a single tuning target at a time.

Next, we randomly combine multiple tuning targets in an attempt to stack objective savings. Specifically, to tune an application, we create a Random Tuning Profile (RTP) where we randomly select a collection from a range of tuning targets that individually showed successful improvements for this application during the initial search. We then randomly select a fitted Prior for each of the selected tuning targets (a single tuning target can have more than one fitting Prior). We instrument the compiler according to the targets in the RTP and use it to compile the targeted application. Every time an instrumented function is executed, we randomly select an output value from the corresponding Prior's value range. Using this technique, we can boost some binary size savings beyond our results for single tuning targets only.

Figure 9 shows the binary size improvements we achieve when randomly combining the tuning targets for the Full Evaluation Set compared to our results for single tuning targets from Figure 6.2. For some applications like N-cg and P-cor we managed to gain significant improvements of up to 19.5% over the highest savings observed for single tuning targets only during the heuristic search. For others, we found no improvements or stayed below our previous results. Due to limited resource availability, the random search experiment we ran only considered roughly 6400 search points per applications which is a small number compared to the large search space we looked through. On our cluster this took roughly 2-3 hours per application. Nevertheless, the results show that Heureka's suggested heuristics and value ranges can be used by auto-tuning tools to improve optimization objectives. Additionally, we demonstrate that the effects of multiple tuning targets can be stacked to boost binary size savings beyond what is possible with individual heuristics alone.

VI. Related Work

No prior work has ever attempted to automatically discover and exploit heuristics in systems software. Ours is the first to do so. Tuning heuristics, in particular for compilers, is a rich field targeted by many authors [5]–[7], [9], [29]–[33]. Using machine learning models to do so is becoming more and more popular among compiler developers as reviewed recently by Wang and O'Boyle [34] and Leather and Cummins [35].

Trofin et al. [5] introduce a framework to replace heuristics in compilers with a reinforcement learning (RL) approach. They evaluate their method against a case study in which they target inlining heuristics of the LLVM framework. They achieve an average of 7% improvement over an `-Oz` baseline. Like Trofin, Haj-Ali et al. [9] use RL to optimize loop vectorisation heuristics to improve the existing cost model used in LLVM. Stephenson et al. [29] identify priority or cost functions as a crucial part of individual compiler optimizations. They propose a genetic programming approach which automatically optimizes a set of compiler heuristics by iteratively searching for priority functions which improve the execution time of generated binaries. Similarly, Cavazos et al. [36] and Kulkarni et al. [7] use genetic algorithms to optimize inlining heuristics for just-in-time compilation in Java virtual machines. Moss et al. [31] and later Mendis et al. [32] replace traditional analytical models with supervised learning techniques and use them to predict the throughput of instructions in a basic block in the instruction selection phase of a compiler. Hollenbeck et al. [37] manually select and parameterize twelve constant values from the Haskell GHC compiler. They modify them by randomly sampling configurations from a uniform and a normal distribution resembling our Priors to improve the inlining optimization. All of the

studies above show improvements for the compiler heuristics they have chosen to tune or replace with an ML approach but their selection of which heuristics to target in the first place is done manually. Our method, on the other hand, is able to find and parameterize high value targets automatically, even non-obvious ones.

Differential Testing [17] is an efficient technique to automatically process and evaluate a large number of test cases against different versions of the same program or similar programs. It has been successfully used by multiple research studies on exposing bugs in compilers to drive large testing efforts [38]–[41]. In a compiler fuzzing study, Yang et al. [41] employ a Differential Testing approach to expose compiler bugs. They automatically generate a large number of tests by sampling a probabilistic grammar which covers a subset of the C programming language. Cummins et al. [40] later improve upon their test generation method by using a deep learning model which they train on hand written code mined from open source GitHub repositories. Similar to our study, they also make use of Differential Testing techniques to automatically evaluate the test outcome of their generated program code.

VII. Conclusion and Future Work

In this study we introduce Heureka, a framework to fully automatically search through the LLVM compiler and find heuristics encapsulated into functions. Using a Differential Testing approach, we fit function image templates, aka. Priors, to approximate the output value range of targeted heuristic functions. If we can fit a Prior to a function's output channel and observe at least one value that shows objective improvements, we suggest the output channel as a tuning target. With Heureka we evaluate eight thousand functions from the LLVM OPT source code and identify tuning targets to serve two use cases: Firstly, to tune individual applications and secondly to find promising heuristics in the compiler for further manual examination and optimization. We heavily tune the binary size of single applications and achieve 7.7% reduction on average compared to an `-Oz` baseline for single tuning targets and up to 19.5% when combining the effects of multiple targets using a random search. Secondly, we report heuristic entries from the strictest tuning target subset which generalize well and show high improvements for most of the evaluated applications. They are intended to be picked up by compiler developers for evaluation and integration.

For our future work, we plan to introduce more sophisticated Priors capable of picking up on more complex heuristics by considering output and input values. Also, our current Priors aim at individual output channels and do not yet consider interdependences between them. In this work, we have focused on reducing the binary size of generated binaries. Heureka can, however, target different optimization objectives like application runtime. We plan to evaluate benchmarks for runtime and combinations of runtime and binary size. We want to expand upon our initial approach of stacking individual tuning targets by replacing the random search with a more powerful technique, such as reinforcement learning. Heureka can be adapted to suggest heuristics for any program and desired optimization goal as long as a range of test cases is available. We plan to target other systems to uncover potential heuristics, like operating systems or SQL engines. Lastly, in this study, we target only heuristics encapsulated into functions which is certainly not always the case. Different call sites of the same function might benefit in different ways from heuristic output and can be treated as different heuristics. An even more fine-grained approach would be to consider individual assignments as heuristic outputs. Every time a value is assigned in the code, we can potentially hook in and tune it. We intend to tackle this generalized approach in our future work.

References

[1] A. F. d. Silva, B. N. B. de Lima, and F. M. Q. Pereira, "Exploring the space of optimization sequences for code-size reduction: insights and tools," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2021. New York, NY, USA: Association for Computing Machinery, Feb. 2021, pp. 47–58. [Online]. Available: https://doi.org/10.1145/3446804.3446849

[2] G. G. Fursin, M. F. P. O'Boyle, and P. M. W. Knijnenburg, "Evaluating Iterative Compilation," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, B. Pugh and C.-W. Tseng, Eds. Berlin, Heidelberg: Springer, 2005, pp. 362–376.

[3] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August, "Compiler optimization-space exploration," in *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, Mar. 2003, pp. 204–215.

[4] K. D. Cooper, D. Subramanian, and L. Torczon, "Adaptive Optimizing Compilers for the 21st Century," *The Journal of Supercomputing*, vol. 23, no. 1, pp. 7–22, Aug. 2002. [Online]. Available: https://doi.org/10.1023/A:1015729001611

[5] M. Trofin, Y. Qian, E. Brevdo, Z. Lin, K. Choromanski, and D. Li, "MLGO: a Machine Learning Guided Compiler Optimizations Framework," *arXiv:2101.04808 [cs]*, Jan. 2021, arXiv: 2101.04808. [Online]. Available: http://arxiv.org/abs/2101.04808

[6] K. Hoste and L. Eeckhout, "Cole: compiler optimization level exploration," in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '08. New York, NY, USA: Association for Computing Machinery, Apr. 2008, pp. 165–174. [Online]. Available: https://doi.org/10.1145/1356058.1356080

[7] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon, "Automatic construction of inlining heuristics using machine learning," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2013, pp. 1–12.

[8] M. Stephenson and S. Amarasinghe, "Predicting unroll factors using supervised classification," in *International Symposium on Code Generation and Optimization*, Mar. 2005, pp. 123–134.

[9] A. Haj-Ali, N. K. Ahmed, T. Willke, Y. S. Shao, K. Asanovic, and I. Stoica, "NeuroVectorizer: end-to-end vectorization with deep reinforcement learning," in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, Feb. 2020, pp. 242–255. [Online]. Available: https://doi.org/10.1145/3368826.3377928

[10] LLVM, "Llvm source project," 2022. [Online]. Available: https://github.com/llvm/llvm-project

[11] D. H. Bailey, "NAS Parallel Benchmarks," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 1254–1259. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_133

[12] L.-N. Pouchet and S. Grauer-Gray, "Polybench: The polyhedral benchmark suite.(2012)," 2012. [Online]. Available: http://web.cs.ucla.edu/~pouchet/software/polybench/

[13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY,

USA: Association for Computing Machinery, Jun. 2013, pp. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[14] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: an extensible framework for program autotuning," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, Aug. 2014, pp. 303–316. [Online]. Available: https://doi.org/10.1145/2628071.2628092

[15] W. F. Ogilvie, P. Petoumenos, Z. Wang, and H. Leather, "Minimizing the cost of iterative compilation with active learning," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2017, pp. 245–256.

[16] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," Jun. 2018, arXiv:1802.04730 [cs]. [Online]. Available: http://arxiv.org/abs/1802.04730

[17] W. M. McKeeman, "Differential Testing for Software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[18] G. Alvarez, "App size matters I," May 2020, section: Technology. [Online]. Available: https://www.farfetchtechblog.com/en/blog/post/app-size-matters-i/

[19] K. Beladiya, "9 Ways to Reduce Android App Size During Android App Development," Jan. 2022. [Online]. Available: https://theonetechnologies.com/blog/post/5-ways-to-reduce-android-app-size

[20] S. Tolomei, "Shrinking APKs, growing installs," Nov. 2017. [Online]. Available: https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2

[21] E. Camber, "Size matters: How your app size is costing you customers," Nov. 2018. [Online]. Available: https://medium.com/pixplicity/size-matters-how-your-app-size-is-costing-you-customers-6121d6db74e

[22] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *ACM SIGPLAN Notices*, vol. 44, no. 6, pp. 177–187, Jun. 2009. [Online]. Available: https://doi.org/10.1145/1543135.1542496

[23] A. Maramzin, C. Vasiladiotis, R. C. Lozano, M. Cole, and B. Franke, ""It looks like you're writing a parallel loop": a machine learning based parallelization assistant," in *Proceedings of the 6th ACM SIGPLAN International Workshop on AI-Inspired and Empirical Methods for Software Engineering on Parallel Computing Systems*, ser. AI-SEPS 2019. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3358500.3361567

[24] LLVM, "LLVM's Analysis and Transform Passes — LLVM 13 documentation," 2022. [Online]. Available: https://llvm.org/docs/Passes.html

[25] ——, "llvm-size - print size information — LLVM 13 documentation," 2022. [Online]. Available: https://llvm.org/docs/CommandGuide/llvm-size.html

[26] M. Liška, "C-Vise: Super-parallel python port of c-reduce," Aug. 2022. [Online]. Available: https://github.com/marxin/cvise

[27] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for C compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: Association for Computing Machinery, Jun. 2012, pp. 335–346. [Online]. Available: https://doi.org/10.1145/2254064.2254104

[28] Altair, "Distributed Resource Management and Optimization | Altair Grid Engine," 2021. [Online]. Available: https://www.altair.com/grid-engine/

[29] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: improving compiler heuristics with machine learning," *ACM SIGPLAN Notices*, vol. 38, no. 5, pp. 77–90, May 2003. [Online]. Available: https://doi.org/10.1145/780822.781141

[30] C. Cummins, P. Petoumenos, Z. Wang, and H. Leather, "End-to-End Deep Learning of Optimization Heuristics," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 219–232.

[31] J. E. B. Moss, P. E. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. E. Brodley, and D. Scheeff, "Learning to schedule straight-line code," in *NIPS*, 1997, pp. 929–935. [Online]. Available: http://papers.nips.cc/paper/1349-learning-to-schedule-straight-line-code

[32] C. Mendis, A. Renda, S. Amarasinghe, and M. Carbin, "Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks," *arXiv:1808.07412 [cs, stat]*, May 2019, arXiv: 1808.07412. [Online]. Available: http://arxiv.org/abs/1808.07412

[33] M. Hashemi, K. Swersky, J. Smith, G. Ayers, H. Litz, J. Chang, C. Kozyrakis, and P. Ranganathan, "Learning Memory Access Patterns," in *Proceedings of the 35th International Conference on Machine Learning*. PMLR, Jul. 2018, pp. 1919–1928, iSSN: 2640-3498. [Online]. Available: https://proceedings.mlr.press/v80/hashemi18a.html

[34] Z. Wang and M. O'Boyle, "Machine Learning in Compiler Optimization," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1879–1901, Nov. 2018, conference Name: Proceedings of the IEEE.

[35] H. Leather and C. Cummins, "Machine Learning in Compilers: Past, Present and Future," in *2020 Forum for Specification and Design Languages (FDL)*, Sep. 2020, pp. 1–8, iSSN: 1636-9874.

[36] J. Cavazos and M. O'Boyle, "Automatic Tuning of Inlining Heuristics," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Nov. 2005, pp. 14–14.

[37] C. Hollenbeck, M. F. P. O'Boyle, and M. Steuwer, "Investigating magic numbers: improving the inlining heuristic in the Glasgow Haskell Compiler," in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, ser. Haskell 2022. New York, NY, USA: Association for Computing Machinery, Sep. 2022, pp. 81–94. [Online]. Available: https://doi.org/10.1145/3546189.3549918

[38] F. Sheridan, "Practical testing of a C99 compiler using output comparison," *Software: Practice and Experience*, vol. 37, no. 14, pp. 1475–1488, 2007, _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.812. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.812

[39] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *Proceedings of the 8th ACM international conference on Embedded software*, ser. EMSOFT '08. New York, NY, USA: Association for Computing Machinery, Oct. 2008, pp. 255–264. [Online]. Available: https://doi.org/10.1145/1450058.1450093

[40] C. Cummins, P. Petoumenos, A. Murray, and H. Leather, "Compiler fuzzing through deep learning," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, Jul. 2018, pp. 95–105. [Online]. Available: https://doi.org/10.1145/3213846.3213848

[41] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

APPENDIX A
FUNCTION INSTRUMENTATION

```
1 int add(int a, int b) {
2     return a + b;
3 }
```

10.1: Original unmodified function code.

```
1 ...
2 static int (*add__fn__)(int, int) =
      add__original__;
3 static FnExtensionPoint* add__extension_point__;
4
5 static int add__original__(int a, int b) {
6     return a + b;
7 }
8
9 static int add__extended__(int a, int b) {
10    int r;
11    void* args[] = { &a, &b };
12    Internal::eval(add__extension_point__, &r,
      args);
13    return r;
14 }
15
16 static void add__reflect__(void* r_val, void*
      arg_vals[]){
17    int* rp = reinterpret_cast<int*>(r_val);
18    int* ap = reinterpret_cast<int*>(arg_vals[0]);
19    int* bp = reinterpret_cast<int*>(arg_vals[1]);
20    *rp = add__original__(*ap, *bp);
21 }
22
23 int add(int a, int b) {
24    return add__fn__(a,b);
25 }
26
27 __attribute__((constructor))
28 void add__init__() {
29    ...
30    add__extension_point__ =
31        Internal::create_extension_point(...);
32 }
```

10.2: Explicit modification of the original `add` function in C++. The instrumenter replaces the original function body with a call to a registered function pointer (`add__fn__`) which points to either a copy of the original function (`__original__`) or a replacement function containing custom extension code (`__extended__`). An additional function is added to allow calling the original function from extension code (`__reflect__`). Lastly, extension point registration is done in a global constructor (`__init__`).

Fig. 10: Example instrumentation of `add` function showing explicit instrumentation code in C++ to illustrate what is being generated in LLVM-IR.

At the core of our evaluation method is the Differential Testing of compiler functions for various outputs. This requires the ability to modify the output values of arbitrary functions in the compiler. To instrument functions within the LLVM compiler targeted by Heureka, we build it from source using an unmodified vanilla LLVM compiler. Our vanilla compiler runs a pass which picks up target functions during compilation of the target compiler and adds required instrumentation.

The added instrumentation is designed to be as light weight as possible to minimise the impact on compiler execution. In its basic form, it offers no more than an extension point per target function. These extension points are ignored if no extension code with added functionality is provided as dynamic library and translate into a single pointer dereference for an indirect call as overhead over the uninstrumented version of a target function.

We built an extension library which uses aspect oriented programming to insert additional functionality before, after or around an instrumented target function. The C++ extension code is loaded dynamically when using the instrumented compiler with the flags `-Xclang -load -Xclang /path/to/libextension.so`. The extension library gives extension code access to the original input parameters of the target function and allows calling the target function so that original output parameters can be recorded and modified as well.

Figure 10 shows an example instrumentation of a simple `add` function which adds two integer numbers together. Figure 10.1 shows the C++ code for the original unmodified `add` function while Figure 10.2 shows the modified version. The actual modification is done in LLVM-IR which is much more verbose and less intuitive to understand than high level code. Hence, for illustration purposes, Figure 10.2 shows what a translation to explicit C++ code would look like.

The instrumenter replaces the body of the original function with a single function pointer dereference and a call to the registered pointer target (`add__fn__`). If no extension code is provided, a copy of the original function is registered (`__original__`) as pointer target. Otherwise, an extension function is called which wraps around the extension code (`__extended__`). All extension code can be accessed via a `FnExtensionPoint` object which is created for each extended function. To call the original function from the extension code, a reflection function is offered as well (`__reflect__`). Extension point object creation and initialisation with the required function type is done in a global constructor which is generated for each instrumented function (`__init__`). Our function instrumentation pass is scheduled as early as possible before other optimizations such as inlining are applied.

Once extension points are added to one or more functions in the compiler, Heureka offers functionality to replace existing function code with custom extensions. Extension libraries are loaded dynamically at runtime of the target software and replace instrumented original functions as required.

Figure 11 shows example extension code using the C++ and Python APIs of our auto tuning extension library. In both cases, the `add` function is extended using an after advice hook, i.e. the extension code is called after the original function was used in the target software. This way, the return value of the original function can be modified. Here, it is incremented by one and returned to the calling code.

The auto tuning extension library uses an internal type system comparable to LLVM's type system to identify function prototypes and corresponding parameter and return value types. Next to registering after advice listeners, before advice

and around advice aspects can be used to hook into an extension point of one or more instrumented functions. There are also some lower level methods available where the function pointer of the extension code is set explicitly. If struct type hierarchies are needed for target function signatures and output channels, they are recorded during instrumentation by the compiler pass and generated for the extension code as required.

Instrumentation pass and extension library are not restricted to instrumenting the LLVM compiler but can be used for arbitrary programs as long as the source code is available. We published instrumentation pass and extension library in our online repository at *redacted-for-review*.

```
1  struct AddOneAfterListener: Listener {
2      AfterAdvice add_one = [this](FnExtensionPoint&
        pt, RetVal ret_value, ArgVals arg_values) {
3          const TypeDesc* type = pt.get_return_type
       ();
4          if (type == IntTypeDesc::get_i32()) {
5              int* r = reinterpret_cast<int*>(
       ret_value);
6              *r += 1;
7          }
8      };
9      void on_extension_point_register(
       FnExtensionPoint& pt) {
10         if (pt.get_return_type()->
       get_discriminator() == TypeDesc::INT) {
11             pt.extend_after(add_one, id);
12         }
13     }
14     void on_extension_point_unregister(
       FnExtensionPoint& pt) {
15         pt.remove_after(id);
16     }
17     AdviceId id = get_unique_advice_id();
18 };
19 ListenerLifeCycle<AddOneAfterListener>
       addOneAfterListener;
```

11.1: Example of extension code for the `add` function written in C++. The extension is applied to all extension points of functions with 32-bit integer return values (line 10-11) and adds one to the returned result after the original function call (line 3-6).

```
1 def incr_ret(pt, ret, *args):
2     ret.value += 1
3
4 auto2tune.extend_after(incr_ret, name_pred = "
      _Z3addii")
```

11.2: Example extension code written using the Python interface. This extension is explicitly applied to the `add` function extension point which is identified by its mangled name (line 4). It also adds one to the result of the original function (line 2).

Fig. 11: Example extension code for the `add` function using C++ and Python interfaces to add one to the result of the original function call.

APPENDIX B
SELECTION OF HEURISTIC TUNING TARGETS

This Section lists a selection of heuristic tuning targets we identified during our experiments. They all generalize well for the tested applications and positively affect binary size for many of them if tuned. Also, they are safe to use within given Prior value ranges for all applications, i.e. compilation or application binaries work as expected and do not crash if tuned. We list them here to exemplify heuristics found by Heureka and discuss examples in detail in Section V-D.

A heuristic reported by Heureka is specified by its function name, the LLVM module where it was picked up during instrumentation and the heuristic output channel for which Heureka could find fitting Priors. We also list the full function signature and source location as found in the LLVM source code version 10.0.1, as well as fitted Priors with their value ranges. A Prior's value range can include all possible values of the output channel's type or a given upper and lower bound (see Section III).

A. Function List

Function Name	isNoAliasArgument
LLVM Module	`llvm/lib/Analysis/AliasAnalysis.cpp`
Source Location	`llvm/lib/Analysis/AliasAnalysis.cpp` line 888
Output Channel	return value of type `bool`
Signature	bool isNoAliasArgument(const Value *V)
Priors	Boolean Prior [complete type range]
Description	Affects memory access code by modifying noalias attribute.

Function Name	alias
LLVM Module	`llvm/lib/Analysis/AliasAnalysis.cpp`
Source Location	`llvm/lib/Analysis/AliasAnalysis.cpp` line 105
Output Channel	enum return value of type `AliasResult`
Signature	AliasResult AAResults::alias(const MemoryLocation &LocA, const MemoryLocation &LocB)
Priors	Integer Offset Prior [0,1] Integer Scale Prior [1,2]
Description	Affects alias analysis result and whether pointers are aliased to each other.

Function Name	FindFunctionBackedges
LLVM Module	`llvm/lib/Analysis/CFG.cpp`
Source Location	`llvm/lib/Analysis/CFG.cpp` line 27
Output Channel	function argument `Result->Size` of type `unsigned`
Signature	void FindFunctionBackedges(const Function &F, SmallVectorImpl¡std::pair¡const BasicBlock*, const BasicBlock*¿¿ & Result)
Priors	All Integers Prior [complete type range]
Description	Reducing the number of loop headers found in a function to zero can reduce binary size.

Function Name	getModRefInfoForArgument
LLVM Module	`llvm/lib/Analysis/GlobalsModRef.cpp`
Source Location	`llvm/lib/Analysis/GlobalsModRef.cpp` line 896
Output Channel	enum return value of type `ModRefInfo`
Signature	ModRefInfo GlobalsAAResult::getModRefInfoForArgument(const CallBase *Call, const GlobalValue *GV, AAQueryInfo &AAQI)
Priors	All Integers Prior [complete type range]
Description	Affects information on how function arguments accesses memory.

Function Name	isLoopSimplifyForm
LLVM Module	`llvm/lib/Analysis/LoopInfo.cpp`
Source Location	`llvm/lib/Analysis/LoopInfo.cpp` line 465
Output Channel	return value of type `bool`
Signature	bool Loop::isLoopSimplifyForm() const
Priors	Boolean Prior [complete type range]
Description	Decides if a loop is in simplified form and can be optimised.

Function Name	hasDedicatedExits
LLVM Module	`llvm/lib/Analysis/LoopInfo.cpp`
Source Location	`llvm/include/llvm/Analysis/LoopInfoImpl.h` line 85
Output Channel	return value of type `bool`
Signature	bool LoopBase¡llvm::BasicBlock, llvm::Loop¿::hasDedicatedExits() const
Priors	Boolean Prior [complete type range]
Description	Decides if exit blocks for a loop have predecessors outside the loop and affects loop optimization.

Function Name	getBlockingAccess
LLVM Module	`llvm/lib/Analysis/MemorySSA.cpp`
Source Location	`llvm/lib/Analysis/MemorySSA.cpp` line 623
Output Channel	return value field `TerminatedPath.LastNode` of type `unsigned`
Signature	Optional¡TerminatedPath¿ getBlockingAccess(const MemoryAccess *StopWhere, SmallVectorImpl¡ListIndex¿ &PausedSearches, SmallVectorImpl¡ListIndex¿ &NewPaused, SmallVectorImpl¡TerminatedPath¿ &Terminated)
Priors	Integer Scale Prior [0,1]
Description	Affects optimization of phi nodes.

Function Name	getLoopDisposition
LLVM Module	`llvm/lib/Analysis/ScalarEvolution.cpp`
Source Location	`llvm/lib/Analysis/ScalarEvolution.cpp` line 11798
Output Channel	enum return value of type `LoopDisposition`
Signature	ScalarEvolution::LoopDisposition ScalarEvolution::getLoopDisposition (const SCEV *S, const Loop *L)
Priors	Integer Scale Prior [0,2147483649]
Description	Affects loop optimization for scalar evolutions.

Function Name	sanitizeFunctionName
LLVM Module	`llvm/lib/Analysis/TargetLibraryInfo.cpp`
Source Location	`llvm/lib/Analysis/TargetLibraryInfo.cpp` line 609
Output Channel	return value field `StringRef.Length` of type `size_t`
Signature	static StringRef sanitizeFunctionName(llvm::StringRef funcName)
Priors	Integer Offset Prior [-1,0]
Description	Affects optimization of built-in functions.

Function Name	dropLLVMManglingEscape
LLVM Module	`llvm/lib/Analysis/TargetLibraryInfo.cpp`
Source Location	`llvm/include/llvm/IR/GlobalValue.h` line 481
Output Channel	return value field `StringRef.Length` of type `size_t`
Signature	static StringRef dropLLVMManglingEscape(StringRef Name)
Priors	Integer Offset Prior [-1,0]
Description	Affects optimization of built-in functions.

Function Name	mustSuppressSpeculation
LLVM Module	`llvm/lib/Analysis/ValueTracking.cpp`
Source Location	`llvm/lib/Analysis/ValueTracking.cpp` line
Output Channel	return value of type `bool`
Signature	bool llvm::mustSuppressSpeculation(const LoadInst &LI)
Priors	Boolean Prior [complete type range]
Description	Affects optimization of load instructions.

Function Name	RemoveOne
LLVM Module	`llvm/lib/Transforms/InstCombine/InstructionCombining.cpp`
Source Location	`llvm/include/llvm/Transforms/InstCombine/InstCombineWorklist.h` line 82
Output Channel	object field `this->Worklist.Size` of type `unsigned`
Signature	Instruction* InstCombineWorklist::RemoveOne()
Priors	Integer Scale Prior [0,1]
Description	Affects instruction combiner optimization by modifying its work list.

Function Name	CFGSimplifyPass
LLVM Module	`llvm/lib/Transforms/Scalar/SimplifyCFGPass.cpp`
Source Location	`llvm/lib/Transforms/Scalar/SimplifyCFGPass.cpp` line 250
Output Channel	object field `this->Options.BonusInstThreshold` of type `int`
Signature	CFGSimplifyPass::CFGSimplifyPass(unsigned Threshold = 1, bool ForwardSwitchCond = false, bool ConvertSwitch = false, bool KeepLoops = true, bool SinkCommon = false, std::function¡bool(const Function &)¿ Ftor = nullptr)
Priors	All Integers Prior [complete type range]
Description	Affects folding basic block combinations.

Function Name	analyzeGlobal
LLVM Module	`llvm/lib/Transforms/Utils/GlobalStatus.cpp`
Source Location	`llvm/lib/Transforms/Utils/GlobalStatus.cpp` line 191
Output Channel	function enum argument `GS.Size.StoredType` of type `StoredType`
Signature	static bool analyzeGlobal(const Value *V, GlobalStatus &GS)
Priors	Integer Offset Prior [0,2147483644] Integer Scale Prior [1,715827882]
Description	Affects global value optimization.

Function Name	isNoBuiltin
LLVM Module	`llvm/lib/Transforms/Utils/SimplifyLibCalls.cpp`
Source Location	`llvm/include/llvm/IR/InstrTypes.h` line 1647
Output Channel	return value of type `bool`
Signature	bool CallBase::isNoBuiltin() const
Priors	Boolean Prior [complete type range]
Description	Affects optimization of built-in functions.