Investigating Magic Numbers: Improving the Inlining Heuristic in the Glasgow Haskell Compiler

Celeste Hollenbeck
c.hollenbeck@sms.ed.ac.uk
University of Edinburgh
Edinburgh, UK

Michael F. P. O'Boyle
mob@inf.ed.ac.uk
University of Edinburgh
Edinburgh, UK

Michel Steuwer
michel.steuwer@ed.ac.uk
University of Edinburgh
Edinburgh, UK

Abstract

Inlining is a widely studied compiler optimization that is particularly important for functional languages such as Haskell and OCaml. The Glasgow Haskell Compiler (GHC) inliner is a heuristic of such complexity, however, that it has not significantly changed for nearly 20 years. It heavily relies on hard-coded numeric constants, or *magic numbers*, based on out-of-date intuition. Dissatisfaction with inlining performance has led to the widespread use of inlining pragmas by programmers.

In this paper, we present an in-depth study of the effect of inlining on performance in functional languages. We specifically focus on the inlining behavior of GHC and present techniques to systematically explore the space of possible *magic number* values, or configurations, and evaluate their performance on a set of real-world benchmarks where inline pragmas are present. Pragmas may slow down individual programs, but on average improve performance by 10%. Searching for the best configuration on a per-program basis increases this performance to an average of 27%. Searching for the best configuration for each program is, however, expensive and unrealistic, requiring repeated compilation and execution. This paper determines a new single configuration that gives a 22% improvement on average across the benchmarks. Finally, we use a simple machine learning model that predicts the best configuration on a per-program basis, giving a 26% average improvement.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Compilers**.

Keywords: Inlining, Haskell, GHC

ACM Reference Format:
Celeste Hollenbeck, Michael F. P. O'Boyle, and Michel Steuwer. 2022. Investigating Magic Numbers: Improving the Inlining Heuristic in the Glasgow Haskell Compiler. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium (Haskell '22), September 15–16, 2022, Ljubljana, Slovenia.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3546189.3549918

1 Introduction

Inlining is a classical compiler optimization that has been studied extensively in imperative languages, such as by Davidson and Holler [1992], Chen et al. [1993], and Cavazos and O'Boyle [2005], as well as in functional languages, for example by Peyton Jones and Marlow [2002]. Inlining of function calls is easy to implement in a compiler: replace the call of a function by an instance of the function's body. However, deciding which function call to inline is not straightforward. Chen et al. showed in 1993 the complex performance considerations of inlining, requiring a balance of the benefits of eliminating function call overhead with the additional required code space and its potential downsides, such as increased pressure on the instruction cache.

Peyton Jones and Marlow [2002] pointed out that inlining is particularly important in functional languages, as it subsumes other optimizations that are performed separately in an imperative setting, such as copy propagation and jump elimination.

In addition, functional programs often contain significantly more functions that need to be considered for inlining, due to the frequent use of anonymous functions (a.k.a., *lambda expressions*) and the cultural encouragement to use function abstractions abundantly. Furthermore, inlining is not restricted to functions but can be performed for every `let`-bound variable. Practical functional programming relies on the ability of optimizing compilers, such as the Glasgow Haskell Compiler (GHC), to aggressively inline function calls and compile away the complex abstractions expressed in user code.

Peyton Jones and Marlow, the primary developers of GHC, call effective inlining *"particularly crucial in getting good performance"*, state that *"it is our experience that the inliner is a lead player in many [performance] improvements"*, and also *"No other single aspect of the compiler has received so much attention"* [2002]. Similarly, Minsky highlights the significance for OCaml, as *"inlining is about more than just function call* *overhead. That's because inlining grows the amount of code that the optimizer can look at at a given point, and that makes other optimizations more effective"* [2016]. While OCaml has improved inlining with a new compiler intermediate representation, maybe surprisingly, the approach of GHC to inlining has not changed significantly in the last 20 years.

If this optimization is so critical to functional programming in general, and GHC's performance in particular—why has it not been re-examined, given the massive hardware changes witnessed in the last 20 years? A likely reason is that its inlining decisions are poorly understood, rely on hardwired constants, and are scattered throughout the compiler. While Peyton Jones and Marlow [2002] describe the overall design choices of the GHC inliner and particular implementation challenges, they avoid discussing the crucial numerical parameters that make up the heuristics that eventually decide to inline or not. The heuristics' complex implementation and their reliance on these numbers makes evaluating and modifying GHC's inlining behaviour difficult.

These hand-coded numerical parameters reflect the GHC developers' "best guess" as to what should be inlined, and they are often accommodated by comments expressing the arbitrary nature of the choices made for their values. This highlights them as *magic numbers*, a term described by Miller et al. [2009], and makes modification challenging. Furthermore, changes to these parameters—and GHC in general—are still performance tested against the `nofib` benchmark suite described by Partain in 1993. The `nofib` suite itself is falling into obsoleteness, as observed by Marlow already over 15 years ago [2005]. Inlining in GHC is thus a compiler optimization that is thought to be highly significant, yet difficult to modify and evaluate.

Dissatisfaction with the performance of GHC's inliner is highlighted by developers' frequent use of pragmas to manually annotate their code in an attempt to coerce GHC to inline specific functions and improve performance. Our investigations revealed that 1 in 5 of Haskell projects uploaded to Hackage, the Haskell community's central package archive, contain manually inserted "inline" compiler pragmas.

In this paper, we systematically study inlining in the context of functional languages. We focus specifically on the performance of GHC's inliner, as GHC is one of the most widely used optimizing functional compilers and known to deliver good performance. While our experimental evaluation is specific to GHC, our methodology and findings are of interest to compiler engineers of other functional languages.

We study inlining across a set of real-world Haskell benchmarks where programmers resorted to the use of pragmas to improve inlining performance. We investigate the influence of the magical number values within GHC's inliner by parameterizing them and automatically exploring the space of possible parameter values. In some cases, we observe significant possible performance gains for well-chosen parameter values on a per-benchmark basis. We are also able to find a single parameter configuration that gives an average performance gain across the benchmarks. Finally, we employ a simple machine learning model that predicts good parameter values and delivers good speedups for an unseen program without the need for excessive compilation and execution.

To summarize, we make the following contributions:

- we present a real-world benchmark suite for evaluating GHC's performance, sourced from popular Hackage open-source packages [Hollenbeck 2022];
- we perform an in-depth experimental analysis of the performance of GHC's inliner across a range of real-world benchmarks;
- we show empirical evidence for the benefits of using automated tuning techniques to improve the performance of the GHC inliner;
- and we demonstrate the benefits of using of a simple predictive model that delivers significant performance.

2 Background

2.1 Inlining in Functional Languages

In functional languages, inlining may simply be described as replacing the use of an identifier in an expression with the identifier's definition. An example in Haskell, originally presented by Peyton Jones and Marlow [2002], is given below:

```
let f = \x -> x*3 in f (a+b) - c
⟹  (a+b)*3 - c
```

Peyton Jones and Marlow [2002] identify three distinct program transformations that collectively perform the inlining for the example above:

1. The *inlining* itself replacing a use of a `let`-bound identifier (here: `f`) by a copy of its definition (here: `\x -> x*3`):
```
let f = \x -> x*3 in f (a+b) - c
⟹  let f = \x -> x*3 in (\x -> x*3) (a+b) - c
```
2. *Dead code elimination* that removes unnecessary `let`-bindings where the bound identifier is not used in the body of the `let`, as it is the case in the example:
```
let f = \x -> x*3 in (\x -> x*3) (a+b) - c
⟹  (\x -> x*3) (a+b) - c
```
3. *β-reduction* transforming a lambda application into a `let`-binding, enabling further inlining:
```
(\x -> x*3) (a+b) - c
⟹  (let x = a+b in x*3) - c
```

To finalize the example, we perform more *inlining* and *dead code elimination* steps:

```
(let x = a+b in x*3)-c
⟹  (let x = a+b in (a+b)*3)-c
⟹  (a+b)*3-c
```

As Haskell is a lazy and pure functional language, *inlining*, *dead code elimination*, and *β-reduction* are always legal transformations that do not alter the program's meaning. *Dead code elimination* and *β-reduction* are easy to implement, as both of them are generally beneficial, whereas deciding when to inline what identifier is challenging. Therefore, GHC performs inlining with careful consideration, despite its heavy reliance on good inlining decisions for further optimization. To determine when inlining may expose further opportunities for optimization, GHC must examine the context in which the inlinee occurs to balance the benefits of inlining with potential negative effects, such as code duplication.

2.2 Overview of the GHC Inlining Heuristic

The logic for GHC's inlining decisions is scattered throughout the codebase. A search for "CoreUnfold" brings up 30 different files in GHC's compiler directory. We thus present a simplified account of the heuristic, depicted in Figure 1.

The `callSiteInline` function (top right of Figure 1) is invoked to determine whether to inline or not. Any inlining decision which requires nontrivial consideration is labeled as a `CoreUnfolding` and passed to the function `tryUnfolding` (middle of Figure 1), which makes a value judgment based upon the estimated size of the callee, its arguments, how it fits within its context, and other interesting attributes. At a highly simplistic level, it calculates the cost and benefit of inlining: if the cost minus benefit is less than a threshold, then it performs inlining.

The calculation happens in this line:

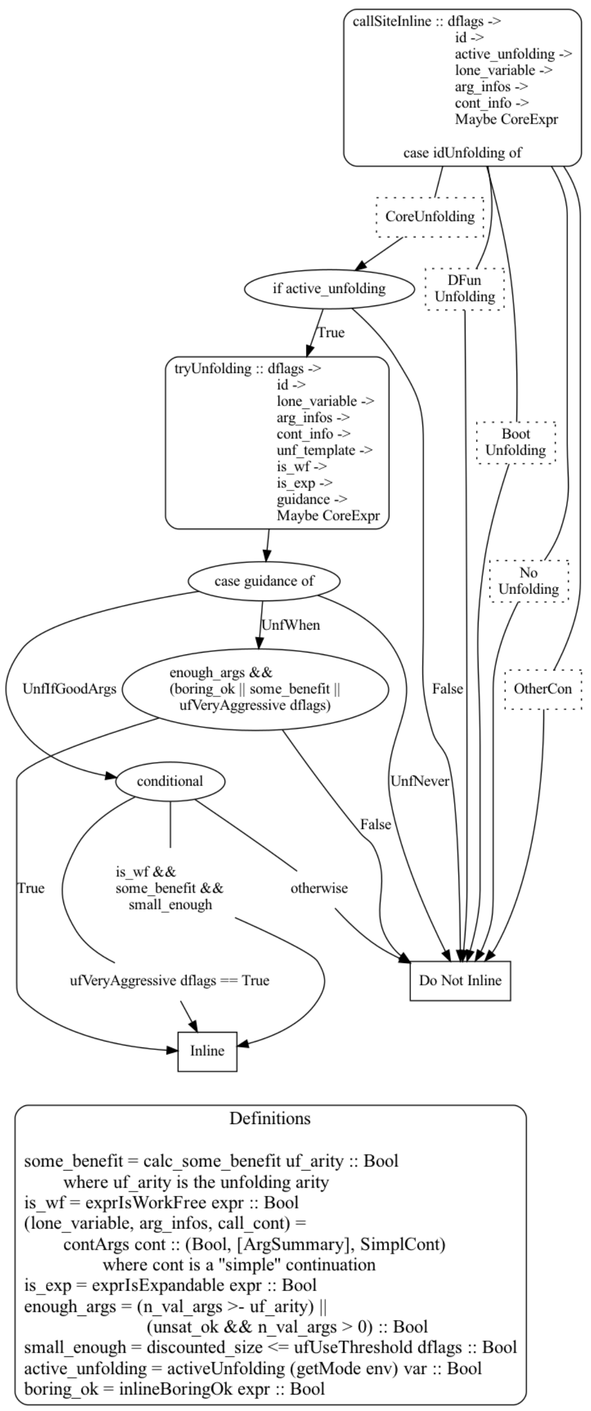
```
small_enough =
    (size - discount) <= ufUseThreshold dflags
```

which determines acceptability, where `size` is determined by a traversal of the inlinee and `discount` is calculated with consideration to the inlinee's arguments, the continuation, and dynamic flags optionally set upon compilation. The discount represents the value gained, which would offset the cost of inlining large things. This computation happens when the Simplifier—a module where GHC iteratively applies optimizations to the Core intermediate representation (Core IR) code—calls `tryUnfolding` on a `CoreUnfolding`.

Each inlining decision additionally depends upon considerations including but not limited to: the type of the expression, its arity, its number and characterization of arguments, the phase of compilation, and a number of calculated discounts and thresholds written directly into GHC simply as best-judgment constants.

2.3 Magic Numbers in the Inliner

The calculations for both `size` and `discount` rely upon several magic numbers written directly into the inliner. An example of the use of these numbers occurs in the first few lines of the function `computeDiscount`, shown in Figure 2, which computes a discount value for all functions being considered for inlining. In `computeDiscount`, the number 10 refers to a discount given for the function itself.


callSiteInline :: dflags -> id -> active_unfolding -> lone_variable -> arg_infos -> cont_info -> Maybe CoreExpr
case idUnfolding of
CoreUnfolding
DFun Unfolding
Boot Unfolding
No Unfolding
OtherCon
if active_unfolding
True
False
tryUnfolding :: dflags -> id -> lone_variable -> arg_infos -> cont_info -> unf_template -> is_wf -> is_exp -> guidance -> Maybe CoreExpr
case guidance of
UnfWhen
UnfIfGoodArgs
UnfNever
enough_args && (boring_ok || some_benefit || ufVeryAggressive dflags)
False
True
conditional
is_wf && some_benefit && small_enough
otherwise
ufVeryAggressive dflags == True
Inline
Do Not Inline
Definitions

```
some_benefit = calc_some_benefit uf_arity :: Bool
        where uf_arity is the unfolding arity
is_wf = exprIsWorkFree expr :: Bool
(lone_variable, arg_infos, call_cont) =
        contArgs cont :: (Bool, [ArgSummary], SimplCont)
                where cont is a "simple" continuation
is_exp = exprIsExpandable expr :: Bool
enough_args = (n_val_args >- uf_arity) ||
                        (unsat_ok && n_val_args > 0) :: Bool
small_enough = discounted_size <= ufUseThreshold dflags :: Bool
active_unfolding = activeUnfolding (getMode env) var :: Bool
boring_ok = inlineBoringOk expr :: Bool
```

Figure 1. Visualization of GHC's Inliner. The function `callSiteInline` is declared in CoreUnfold.hs and is called from Simplify.hs. Rounded boxes indicate functions, ovals indicate conditions, and dotted boxes indicate unfolding IDs.

In this example, making the number 10 larger would give the inlinable item a larger discount to offset its size, increasing its likelihood to be inlined. Such a modification would

```
computeDiscount :: [Int] -> Int -> [ArgSummary]
                         -> CallCtxt
                         -> Int
computeDiscount arg_discounts
                res_discount
                arg_infos
                cont_info
  = 10 -- Discount of 10 because the result
       -- replaces the call
       -- so we count 10 for the function itself
```

Figure 2. First part of the computeDiscount function, with the magic number 10, in CoreUnfold.hs.

make *all* functions more likely to be inlined because they would start with a higher base discount, before the addition of any discounts based upon their arguments (which would be calculated in the lines immediately following in computeDiscount):

```
  + 10 * length actual_arg_discounts
  + round (ufKeenessFactor dflags *
       -- Discount of 10 for each arg supplied,
       -- because the result replaces the call
     fromIntegral (total_arg_discount+res_discount'))
```

Figure 3. Second part of the computeDiscount function, with the magic number 10, in CoreUnfold.hs.

Additionally, the term res_discount', short for "result discount", adds a discount when an efficiency is expected to be gained through inlining—for example, through case reductions. Its numerical value is computed by considering a simplified version of the context, represented by the data type CallCtxt. The original code of Figure 4 shows how some of these possible CallCtxt values are assigned to the magic number 40 to return as a result for res_discount' in one single line of code, along with the comments right after it which debate its accuracy.

Magic numbers such as these are scattered throughout the entire inliner, and its decisions are fundamentally dependent upon them. We set out in this paper to study the impact of these magical numbers systematically.

3 Approach

For this study, we wanted to answer the question: *Could a modification to the inliner's thresholds yield a performance improvement across Haskell code execution time?* If the answer to that question is *yes*, then we face two additional questions: If so, how much improvement might we expect to see by modifying GHC's inlining thresholds? If not, how should we then modify GHC to attain an optimal improvement?

It is necessary to answer these questions before redesigning the inliner, given the complexity of the system. Thus, we constructed a set of benchmarks with the intention of revealing weaknesses in GHC's inlining decision process. We then modified GHC 8.10.3 such that we could change its inliner's magic number values through dynamic flags to see how much we could affect the benchmarks' execution times through the inliner alone.

```
_           -> 40 `min` res_discount
   -- ToDo: this 40 `min` res_discount doesn't
   -- seem right
   --   for DiscArgCtxt it shouldn't matter because
   --     the function will get the arg discount
   --     for any non-triv arg
   --   for RuleArgCtxt we do want to be keener to
   --     inline; but not only constructor results
   --   for RhsCtxt I suppose that exposing a data
   --     con is good in general
   --   And 40 seems very arbitrary
```

Figure 4. GHC 8.10.3: CoreUnfold.hs, line 1640. The top line of code calculates the value for res_discount' seen in Figure 3. The developer's comments highlight some of the arbitrary decisions made.

3.1 Optimization Space Exploration

Because parameterizing all of the inliner's thresholds would have been intractable, we focused on 10 hand-coded magic-number constants to expose as dynamic flags, which could then be passed into GHC when compiling an application. Additionally, in our optimization space, we included two of GHC's built-in dynamic flags. Combined, this totals 12 parameters.

To approximately quantify the type of inlining decisions being performed, we added hooks to GHC 8.10.3 and compiled it against the Cabal library, where Cabal is the canonical system for building and installing Haskell packages. During compilation, GHC performed 8,708,142 nontrivial inlining decisions, where "nontrivial" means any inlining for which it is not obvious that it should definitely be inlined. Among these nontrivial inlinings, 81.8% were designated as UnfIfGoodArgs—which means their unfolding would be large enough to require consideration, but not so large to immediately disqualify it from inlining. Before deciding whether to inline, GHC gives these potential inlinings a reduction in their calculated sizes via a discount calculation:

$$\text{discounted\_size} = \text{size} - \text{discount}\,.$$

We therefore decided to create parameters from magic numbers involved in the calculation of size and discount.

3.2 Characterization of the Parameters

Each parameter was selected because it had a direct impact on GHC's inlining decisions and would likely produce an

Table 1. Inlining parameter dynamic flags, their descriptions, and original values.

Flag	**Description**	**Original Value**
nontrivarg-disc	Discount for an argument labeled "NonTrivial".	10
funcitself-disc	Constant discount value added to every function inlined.	10
actarg-disc	Discount for each argument.	10
discargctxt-disc	Context is the argument of a function with non-zero argument discount.	40
ruleargctxt-disc	Context is the argument of a function with rules.	40
rhsctxt-disc	The context is the right-hand side of a let.	40
arbctxt-disc	The wild card remaining to catch any other type of context and calculate its discount.	40
cosbase	Base size value of a class op.	20
cosargs	Size metric added for each argument of a class op.	10
bigalt	Size component of the biggest alternative when scrutinizing a case expression argument.	20
funfolding-fun-discount	Adjust the eagerness of GHC to inline functions.	60
funfolding-dict-discount	Adjust the eagerness of GHC to inline dictionaries.	30

observable effect on runtime performance. Table 1 describes each parameter and gives their names and original values.

Three parameters, *cosbase*, *cosargs*, and *bigalt*, calculate various components of an inlinee's size. The remaining 9 help calculate its discount, or the numerical value estimated to offset the cost of inlining. We also included the built-in GHC dynamic flags *-funfolding-fun-discount* and *-funfolding-dict-discount*, as they both pertained specifically to inlining.

4 Benchmark Construction

To experimentally evaluate the performance of GHC's inliner, we needed a benchmark suite that would allow us to analyze the performance impact of different inlining decisions.

The `nofib` benchmark suite was originally constructed to be a substantial, diverse, relevant set of programs in 1993; but now, most of its programs run for a fraction of a second, as pointed out by Marlow [2005]. Unfortunately, despite its age, nofib has yet to be replaced or upgraded. As an alternative, we wanted to allow developers to experiment and evaluate on interchangeable, testable, real-world packages from Hackage so that the resultant benchmarks would be heterogeneous and relevant to real-world needs. We based that assumption on previous work to construct a benchmark suite for JavaScript, as described by Richards et al. [2011a].

We therefore constructed a tool in Python to select Hackage packages specifically to suit our benchmarking goal: to identify room for execution time improvement as it pertains to inlining. These programs needed to run for an adequate amount of time, perform a variety of different tasks, and have consistent execution times such that the same inlining decisions would reproduce the same results.

4.1 Benchmark Selection

Stackage is a distribution of a subset of Hackage, where packages within the same snapshot will build together and pass all of their tests. For our benchmarks, we selected packages contained within a single Stackage snapshot.[1] In this Stackage Nightly build, 854 of 2218 packages (about 39%) used QuickCheck, a tool which generates random tests developed by Claessen and Hughes [2000]. Initially, these randomly generated tests were a significant source of unwanted noise. To address this, we set QuickCheck's random seed to one constant and made its test times consistent. We then enabled our scripts to automatically patch all selected packages' dependencies with our modified QuickCheck.

In our Stackage snapshot, 421 of the 2218 packages contained `INLINE` pragmas—or about 19%. We hypothesized these packages may provide code where developers had identified a good set of problems upon which to evaluate inlining. Section 4.2 explains the motivation for that decision.

Influenced by our observation of pragmas, we identified 236 packages with INLINE pragmas in their "src" folders that could be run with `cabal new-test`. From those packages, we sub-selected 10 which each ran over 4 seconds, decreasing the likelihood that any speedup percentages observed would fall outside the range of noise. Table 2 characterizes the selected 10 packages.

4.2 The Consideration of INLINE Pragmas

Compiler pragmas are lines of code specific to individual compilers, rather than the grammars of languages themselves. Programmers insert these pragmas to instruct a compiler on how to process and optimize certain input programs.

In GHC, a pragma to instruct GHC to inline a function is known as an `INLINE` pragma. An `INLINE` pragma may be placed beneath the declaration of a function to coerce GHC to try to inline the function, if it can.

An example of the use of an `INLINE` pragma is:

```haskell
key_function :: Int -> String -> (Bool, Double)
{-# INLINE key_function #-}
```

[1]`stackage-nightly-2020-01-31`

Table 2. Selected Stackage packages and their information. SLOC are estimates. Descriptions were taken from the packages' Hackage profiles.

Package	Version	SLOC	Description	Default Sec.	INLINE Pragmas
hw-rankselect	0.13.3.1	1387	Efficient rank and select operations on large bit-vectors	8.18	88
ListLike	4.6.3	3402	The ListLike package provides typeclasses and instances to allow polymorphism over many common datatypes.	23.04	2
loop	0.3.0	155	Fast loops (for when GHC can't optimize forM_)	19.94	8
metrics	0.4.1.1	1819	High-performance application metric tracking	58.48	9
midi	0.2.2.2	5094	Handling of MIDI messages and files	19.18	2
monoid-subclasses	1.0.1	4900	Subclasses of Monoid	35.12	334
nonempty-containers	0.3.3.0	10055	Non-empty variants of containers data types	4.38	520
poly	0.3.3.0	2040	Haskell library for univariate and multivariate polynomials, backed by Vector.	94.56	57
reinterpret-cast-0.1.0	0.1.0	122	Memory reinterpretation casts for Float/Double and Word32/Word64	26.86	3
set-cover	0.1	2781	Solve exact set cover problems like Sudoku, 8 Queens, Soma Cube, Tetris Cube	15.55	16

An `INLINE` pragma does not guarantee inlining. For example, GHC will not inline a function which breaks the loop of a mutually-recursive group. Coercing an inline may not have a positive effect on performance, and the `INLINE` pragma may have no effect if the function is small enough that GHC would inline it anyway. Developers insert these pragmas at their discretion, usually in the hope to improve the program's performance.

4.3 Pragma Example

User-inserted compiler pragmas may hint that a compiler's optimization decisions could be improved. This snippet from poly contains the INLINE pragma `{-# INLINE integral #-}`:

```
-- | Compute an indefinite integral of a polynomial,
-- setting constant term to zero.
--
-- >>> integral (3 * X^2 + 3) :: UPoly Double
-- 1.0 * X^3 + 3.0 * X
integral :: (Eq a,Fractional a,Vector v (Word,a)) =>
            Poly v a -> Poly v a
integral (Poly xs) = Poly
  $ map (\(p,c) -> (p+1, c/(fromIntegral p + 1))) xs
{-# INLINE integral #-}
```

Here, the function `integral` is overloaded. Without inlining it, `integral` would get passed a dictionary of functions for the possible types of 'a'.

When `integral` is inlined, GHC may see that 'a' has a specific type—for example, float—and then specialize for it. In this way, we can sometimes substitute the retrieval and application of unknown higher-order functions with single machine instructions by telling GHC to inline with pragmas. This makes the resultant code much faster.

5 Experimental Setup

To analyze, explore, and improve the performance of the GHC inliner, we perform an in-depth experimental evaluation on our benchmarks. In all experiments, programs are executed 10 times and average time is reported. The default baseline is the execution time of a package compiled with unmodified GHC 8.10.3 and `INLINE` pragmas disabled. We refer to such execution times as *without pragmas*. If `INLINE` pragmas are enabled, this is referred to as *with pragmas*.

We wanted to explore both parameter values which were likely to yield good performance and also values from a larger range; therefore, we sampled from both a normal distribution and a uniform distribution. Sampling from a normal distribution stays near GHC's original values at the mean and assumes that they are reasonable values. The normal distribution therefore takes μ as the original flag's value and $\sigma = 0.4$. If the generated number was negative, number generation recurred until sampling produced a positive value. We ran 140 configurations randomized in this manner: 70 on the packages with pragmas and 70 without.

We ran additional configurations from a uniform distribution with a lower bound of 0 and an upper bound of $2 * \mathtt{N}$, where N was the default value. For this experiment, we collected 250 configurations without pragmas and 250 with pragmas. The final result contained 640 randomly sampled data points, 320 without pragmas and 320 with pragmas. When we evaluate the performance impact of searching for good configurations, we refer to this as *search*. We ran all benchmarks in isolation on a dedicated server (AMD EPYC 7720P CPU, 256 GB RAM).

6 Experimental Results

We first examine the impact of pragmas and a per-program parameter configuration search on the benchmarks. Then

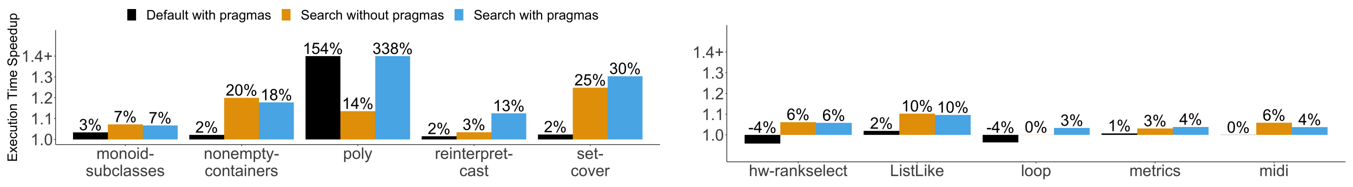


Figure 5. Best-case speedups for each package, grouped by experiment. Speedups are reported as run time ratio along the x-axis and labelled with speedup percentages at the top of bars. The baseline is default GHC without pragmas in package code.

we analyze both the best-performing parameter values, as determined by execution time, and their impact on inlining decisions. Next, we evaluate the performance of the best single fixed-parameter configuration for all programs. We then evaluate and analyze a simple cluster-based model to predict good configurations for an unseen program. Finally, we examine to what extent inlining is architecture-dependent.

6.1 Performance Improvement

Figure 5 shows the best performance improvements found when adding pragmas and configuration search on a per-program basis. All best configurations reported in this figure came from the uniform distribution. The respective geometric means across all programs are summarised in Figure 6.

Pragmas. "Default with pragmas" in Figure 5 shows the performance difference when INLINE pragmas are included in the code versus the baseline (code with pragmas removed). The geometric mean speedup for this experiment was 10%, as shown in Figure 6. However, most of this improvement came from one package, poly, which had a 154% speedup. Although six of the other packages had a speedup above zero, only one had a speedup above 3%. In fact, two packages had negative speedups at -4%, where the pragmas actually had

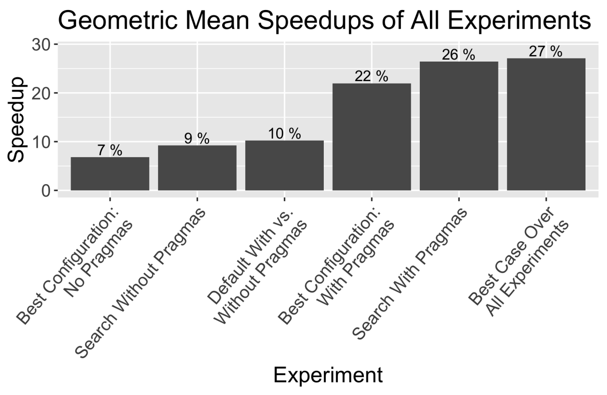


Figure 6. Geometric mean speedups for all experiments. Baseline: Default, without pragmas. "Search With Pragmas" and "Search Without Pragmas" show geometric mean speedups of the averaged best configurations for each package. "Best Configuration" experiments represent the single best configuration applied to all 10 packages.

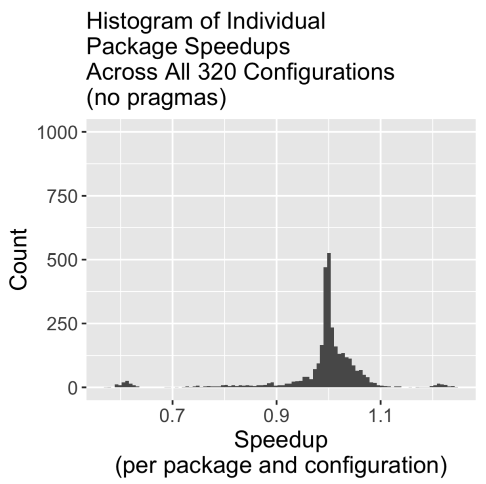
a detrimental effect on execution time. While pragmas can improve the performance of Haskell code, their use shows a variance of results. This may have multiple explanations, three of which are 1) developers may insert pragmas where GHC would normally inline anyway, 2) the effectiveness of the pragmas has changed over time or architectures, or 3) the effectiveness of the pragmas is not verified by or reflected in test cases.

Search. If we search and evaluate configurations in our space on each program without pragmas enabled and report the best value, we get the results labeled "Search without pragmas" in Figure 5. If we repeat this search experiment with pragmas enabled, we get the results in Figure 5 labeled "Search with pragmas".

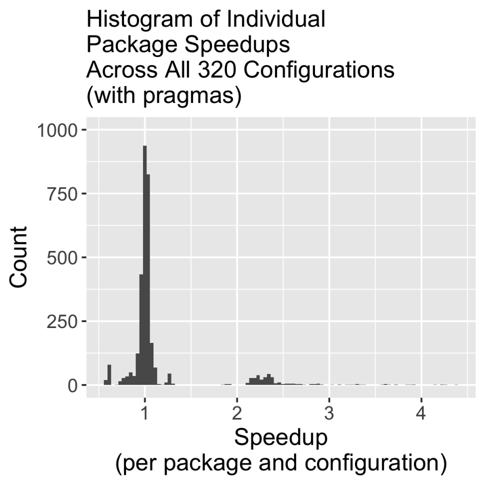
As the results in Figure 6 show, searching for the best configuration on a per-program basis significantly improves performance. Although searching with pragmas gives, on average, better performance than searching without pragmas, Figure 5 shows there are 5 programs where searching without pragmas gives a better individual speedup than the other two experiments: monoid-subclasses, nonempty-containers, hw-rankselect, ListLike, and midi. If we calculate the geometric mean of the best speedups across all experiments, we achieve the maximum possible speedup of 27% shown in Figure 6. Finally, independent of pragmas, searching always delivers a performance improvement. Performance improves without pragmas by 9% and actually has a greater *additional* impact of 16% (26% vs 10%) on packages with pragmas.

Speedup distribution. Figure 7 shows the histograms of speedups achieved with and without pragmas. While the majority of results are clustered around 1 (where 1 indicates no change in execution time), there are some significant positive and negative outliers. For the configurations with pragmas, a speedup over default was observed 64% of the time; and for the configurations with pragmas removed, a speedup was observed 50% of the time. This reflects the earlier observation that inline pragmas, on average, improve performance.

In Figure 8, we examine the best speedup achieved over time as we search the configuration space. The x-axis is the number of configurations evaluated, while the y-axis reports the best speedup achieved so far. There are two lines representing presence of pragmas, solid with pragmas and dotted without pragmas. The baseline is default GHC without



(a)



(b)

Figure 7. Histogram of individual package speedups from search across 320 configurations *without* (a) and *with* (b) package INLINE pragmas.

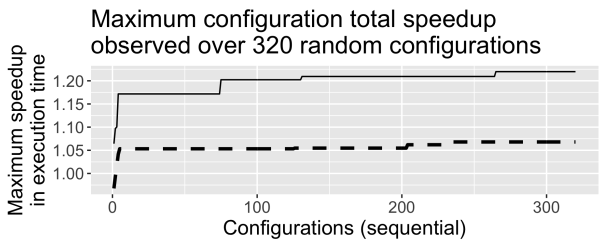
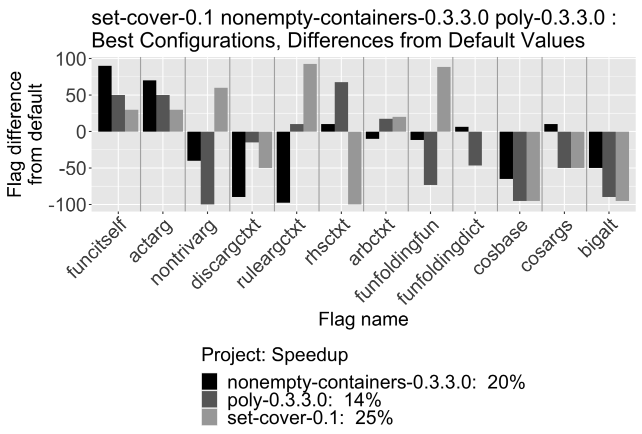
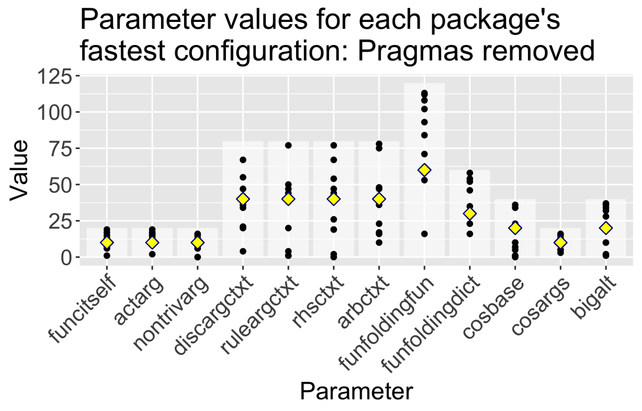


Figure 8. Maximum total speedup of the single best configuration observed over time, search without pragmas (dotted line) and with pragmas (solid).

pragmas. While most speedup is achieved early in the search, many configurations are needed to find the best performance.



(a)



(b)

Figure 9. Top (a): Difference from default for each flag, by top configuration for 3 most improved programs, without pragmas. **Bottom (b)**: Values for each single best configuration for each package. Yellow diamonds are default values. White bars indicate sampling boundaries.

6.2 Analysis

In this section, we examine how the best values of parameters vary across programs.

6.2.1 Parameters.

Distribution. In Figure 9, we took the single best configuration for each package and plotted its values for each parameter. Optimal values for each flag, and for each package, are spread across the range of the random distributions from which they are sampled—with the exception of *funfoldingfun*, which almost entirely prefers a value above 50 (minus one data point).

Looking at a wider set of the most performant configurations, in Figure 10 we filtered the data to include configurations within 1% of the optimal value for each project. We additionally excluded configurations with less than a 3% speedup, which removed all configurations for the packages loop and metrics. In the remaining data, the flag *actarg* seems

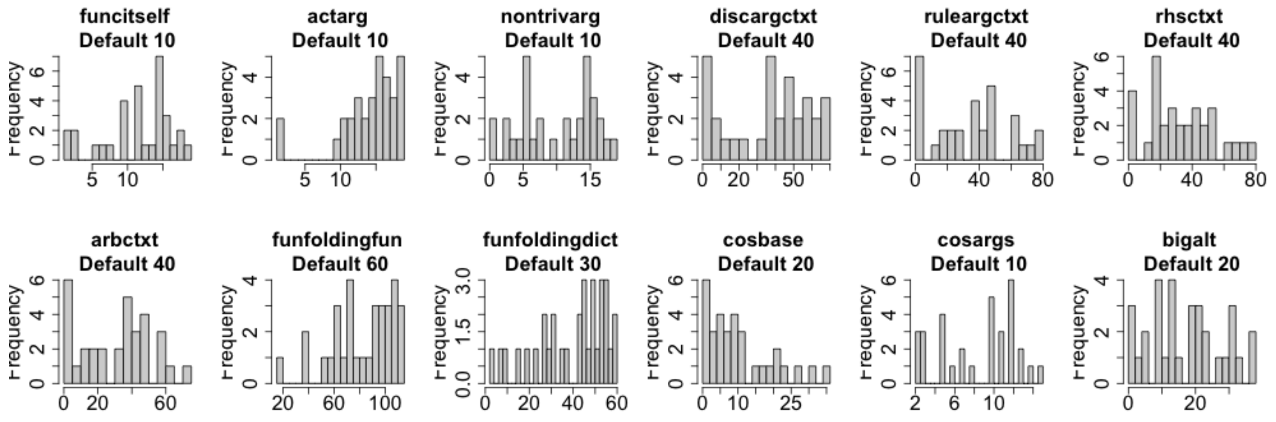


Figure 10. Histograms of parameter values for configurations within 1% of the optimal speedup for each package, across all samplings.

to clearly prefer a value higher than its default (except for 3 configurations in which its value is 0); the flag *cosbase* mostly prefers a value lower than its default; but overall, all of the flags contain values from the lower and upper bounds of their random distributions.

Variation across most-improved programs. If we examine the three best-performing programs in more detail, Figure 9a shows the percent difference from the default values of the best flag configurations for these three packages (without pragmas). The three packages in question, poly, nonempty-containers, and set-cover, have respective speedups of 14%, 20%, and 25%. For almost any flag, the three packages strongly differ—with the flag *arbctxt* having the least disagreement at +20%,-10%, and +17.5%. Where Figure 9b suggests all packages' best-case configurations differ widely on their ideal values, Figure 9a confirms that this disagreement holds even among the three packages with the highest observed speedups.

Reflecting on intuition. We case matched the context parameters *discargctxt*, *ruleargctxt*, *rhsctxt*, and *arbctxt* to collect values to address the comments in Figure 4. Recall the developer deliberated over what the value of the magic number for the call context should be and speculated that the value for `DiscArgCtxt` should not matter, the value for `RuleArgCtxt` should perhaps be higher ("keener to inline"), the value for `RhsCtxt` should probably be high to expose the inlining, and 40 seems rather arbitrary for all of them. The data disagrees with the comments on the two points of: *ruleargctxt* seems to slightly prefer being at or *below* 40, and so does *rhsctxt*. For all four contexts, the range of their values in optimal configurations varies widely.

6.2.2 Inline decisions.

Table 3 shows a comparison of GHC's default inlining behavior to that of the best configuration for each package. To summarize, every best configuration considers more total items for inlining than default GHC, with an averaged 48% increase in number of combined yes and no decisions. Nine of the ten packages decide to inline more total items than default GHC, with the exception of poly. However, all packages inline a smaller percentage of the decisions, relative to total decisions, than default. Additionally, all packages' best configurations decided to inline larger items, with an averaged inlining size of 25.4 versus the default's averaged size of 19.6.

Table 3. Inlining decisions, default vs best.

Package	Total Decisions		% Inlined		Avg. Size Inlined	
	Default	Best	Def.	Best	Def.	Best
hw-rankselect	1,166,781	1,515,164	6.0%	5.7%	14	18
ListLike	910,967	999,237	7.8%	7.4%	15	19
loop	363,940	384,689	6.0%	6.0%	21	25
metrics	355,448	363,286	6.5%	6.4%	19	20
midi	713,076	1,140,791	6.9%	4.8%	26	31
monoid	954,442	1,668,907	5.1%	3.6%	19	29
nonempty	920,901	1,769,534	5.3%	2.8%	26	31
poly	1,405,465	2,635,700	7.1%	3.1%	17	32
reinterpret	361,600	383,446	6.0%	5.8%	20	22
set-cover	414,699	877,758	6.0%	3.1%	19	27

6.2.3 Characterizing Good Inline Decisions.

To better understand the inlining behavior of the default versus best configurations, we collected information vectors for the inlined code in both cases containing information about the compiler intermediate representation (Core IR) at each inlined site, along with the yes or no decision. Summaries of the derived IR vectors are shown in Table 4.

Collection of the Core IR features. To collect features of the inlining decisions, we parsed both the body of the expression being inlined and the calling context, for every nontrivial inlining decision in the benchmarks. Additionally,

Table 4. Collected IR data: their abbreviations, possible values, and descriptions.

Abbr.	Value	Description
Continuation Features		
CSt	Int	Stop[e] = e
CCI	Int	CastIt co K)[e] = K[e ‘cast‘ co]
CAV	Int	(ApplyToVal arg K)[e] = K[e arg]
CAT	Int	(ApplyToTy ty K)[e] = K[e ty]
CSe	Int	(Select alts K)[e] = K[case e of alts]
CSB	Int	(StrictBind x xs b K)[e] = let x = e in K[\xs.b] or equivalently = K[(\x xs.b) e]
CSA	Int	StrictArg (f e1 ..en) K)[e] = K[f e1 .. en e]
Argument Features		
AINT	Int	The argument has structure.
AIVA	Int	The argument is a constructor application, partial application, or constructor-like.
AITA	Int	The argument is not interesting, i.e., deserves no unfolding discount.
ARGS	Int	The number of arguments
Expression Features		
LVAR	Int	Indicates if the expression is a lone variable.
IFV	Int	Number of vars that don't occur in a coercion.
IFJI	Int	Number of join variables
IFL	Int	Number of literals
IFCase	Int	Number of case expressions
IFR	Int	Number of recursive lets
IFLNR	Int	Number of non-recursive lets
IFCast	Int	Number of cast expressions
IFLam	Int	Number of lambda abstractions
IFApp	Int	Number of applications
FOLDTY	{ 0, 1 }	Type of unfolding: UnfWhen or UnfIfGoodArgs
SIZE	Int	The size of the expression
WF	{ 0, 1 }	GHC estimate if expr. will not duplicate work

we included the variables `lone_var` (LVAR); `size` (SIZE); `is_wf` (WF); the type of unfolding (FOLDTY); and the number of arguments to the expression (ARGS). In each decision, we tallied the number of occurrences of expression and continuation Core IR features in a bag-of-words manner, then appended the values for LVAR, WF, FOLDTY, SIZE, and the yes or no inlining decision.

The **Continuation Features** in Table 4 are constructors of the data type `SimplCont` describing a strict context that does not bind any variables. It represents the rest of the expression, above the point of interest, and allows GHC's simplifier to traverse it like a zipper. The inlinee's expression is represented by the features in **Expression Features**, where the values `Var`, `Lit`, `Case`, `Cast`, `Lam`, and `App` are constructors of the recursive data type `CoreExpr`. The features IFR and IFLNR indicate recursive `let` and non-recursive `let`, respectively, which are pattern matched against compositions of the values `Let`, `Rec`, and `NonRec`.

Analysis. We averaged the collected feature vectors across all call sites and programs and compared the difference between the default and best configurations, as shown in Figure 11. For some features there is little difference, such as

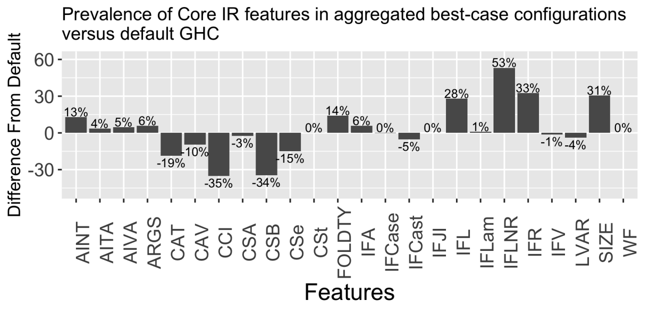


Figure 11. Difference of inlining features, best-case configurations compared against default GHC.

`Case` expressions (IFCase), `Stop` values (CSt), `join` variables (IFJI), and work-free expressions (WF). For other features, there is significant difference—such as far fewer inlinings for `CastIt` (CCI) and `StrictBind` (CSB), and far more inlinings for non-recursive `lets` (IFNLR) and recursive `lets` (IFR). Additionally, the sizes of the best-case inlinings are 31% larger than GHC's default inlinings. As an explanation, non-recursive `let` is an example of an inlining decision associated with anonymous functions particular to functional languages. In summary, then, these features suggest that more inlinings should be performed over anonymous functions, and larger things should be inlined.

6.3 The Single Best Configurations

While searching for good configurations yields better performance, it is is interesting to ask if a single fixed configuration can perform better than the default across all programs. If so, this could replace the existing hard-wired numeric values. Figure 12 shows the speedup achieved when applying the best-found single configuration per program for the packages with and without pragmas, respectively. Table 5 shows these two configurations' parameter values. On average, a mean speedup of 7% is achieved with pragmas disabled, and 22% with pragmas enabled. Thus, when searching for ideal configurations is too expensive, then using a new single best configuration gives already significant improvement.

6.4 A Simple Machine Learning Predictive Model

We saw that when searching for the best configuration per program, we achieve an average speedup of 26%; however, only 22% is achievable with a single, fixed, best-on-average configuration. Next, we investigate whether a simple machine learning approach can improve performance without the need to search many configurations to find an optimum.

We use a simple top-down dynamic programming algorithm for clustering: start with the single configuration that has the most programs at their optimal performance as the initial cluster, then place the poorest performing program into a new cluster within its own best configuration along

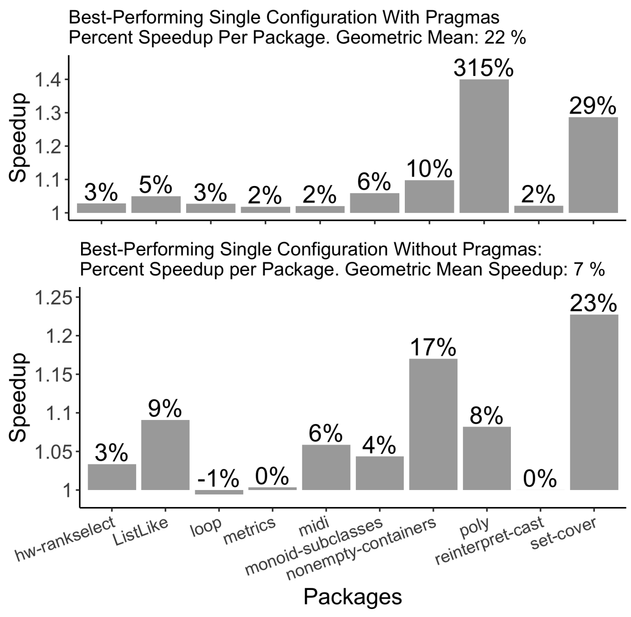


Figure 12. Best-performing single configurations. Top: with pragmas. Bottom: without pragmas. All speedups relative to baseline of unmodified GHC times without pragmas.

Table 5. Parameter values for configuration 229 (single best without pragmas) and 265 (best with pragmas). Default GHC values in rightmost column. 1) funfolding-fun-discount. 2) funfolding-dict-discount.

Parameter	229 (Without Pragmas)	265 (With Pragmas)	GHC
nontrivarg-disc	15	11	10
funcitself-disc	6	1	10
actarg-disc	17	17	10
discargctxt-disc	55	40	40
ruleargctxt-disc	38	60	40
rhsctxt-disc	54	19	40
arbctxt-disc	23	34	40
cosbase	0	22	20
cosargs	15	2	10
bigalt	37	38	20
funfolding-fun[1]	71	115	60
funfolding-dict[2]	58	49	30

with any other program that performs better in that configuration. Recur on this step until just before the number of desired clusters is exceeded.

With a new unseen program, we must determine which cluster it belongs to based on similarity of features. Although static or dynamic program features could be used, we instead use the speedups of the program on a set of selected configurations to determine which cluster it belongs in. Depending on the speedups the program exhibits against each configuration, we allocate it to the cluster of the fastest speedup and

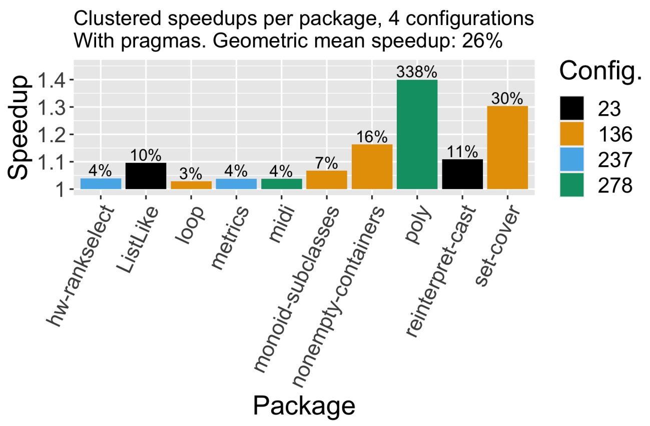


Figure 13. Performance of model. Speedups for each package, using a 4-cluster based predictive model with pragmas.

Table 6. Parameter values for configurations 136, 23, 237, and 278 in Figure 13. Default GHC values shown in rightmost column. 1) funfolding-fun-discount. 2) funfolding-dict-discount.

Parameter	136	23	237	278	GHC
nontrivarg-disc	6	5	6	14	10
funcitself-disc	11	13	7	13	10
actarg-disc	18	16	14	11	10
discargctxt-disc	60	28	29	22	40
ruleargctxt-disc	0	32	13	24	40
rhsctxt-disc	56	17	39	54	40
arbctxt-disc	64	69	32	35	40
cosbase	1	17	38	7	20
cosargs	13	12	15	11	10
bigalt	37	9	38	38	20
funfolding-fun[1]	71	101	77	108	60
funfolding-dict[2]	43	21	17	57	30

assign it that configuration of magic numbers. We performed this learnt approach with leave-one-out cross-validation across the benchmarks.

Analysis. To analyse the results, we focus on a single clustering, where we trained the model without the first benchmark and then assigned it to the best cluster. This is presented in Figure 13, showing an average speedup of 26% across the benchmark suite.

Table 6 shows the values of the parameters in each cluster configuration. It can be highlighted that some parameters are similar across clusters, but seldom in complete agreement. The `cosargs` parameter has the closest values to each other at 13, 12, 15, and 11—also near the default 10. The `funfolding-fun-discount` parameter is consistently higher than default GHC across all four configurations. Three clusters (136, 237, and 278) have very close values of `bigalt` near 38, but cluster 23 prefers this value at 9; and similarly,

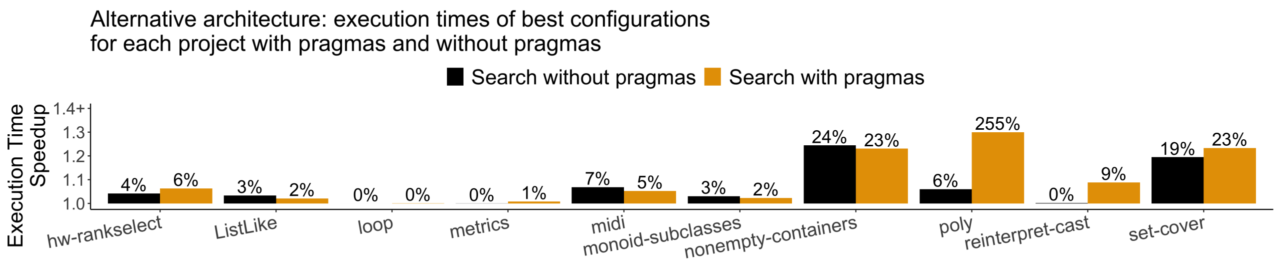


Figure 14. Execution times for best-case configurations for each project on alternative architecture. Geometric mean speedup of configurations without pragmas: 6%. Geometric mean speedup of configurations with pragmas: 21%.

three clusters prefer a `discargctxt-disc` value in the twenties, yet cluster 136 prefers a value of 60. This further supports the possibility that no single configuration will ever approach the optimal improvement observed so far for each type of program.

Towards better inline heuristics. An extended approach would be to build a model that predicts whether to inline or not at each site based on code structure. Core IR vectors could be used as features for a machine-learning-based inlining model and is future work.

6.5 Cross-Architecture Transference

To investigate whether inlining behavior is independent of platform, we evaluated the best performing configurations on a new machine. The second machine was an Intel Xeon CPU E3-1270 v6 with 4 cores running at 3.8 GHz with 62 GB RAM on Debian GNU/Linux 10.

The results are shown in Figure 14. On average, we are able to obtain a 6% and 21% improvement without and with pragmas, respectively. This fares favorably when compared to the original machine where 9% and 26% were respectively found. These encouraging results show that improvements to the inliner may port across machines; however, further work will be needed to confirm this.

6.6 Results Summary

Our experiments have demonstrated that a change of the GHC inliner could yield a significant performance improvement, demonstrably up to 27%, which we attained by modifying the inliner's magic numbers. However, much of that improvement is uncovered with the help of INLINE pragmas. A single new magic number configuration, with help from pragmas, can get us to a 22% improvement—and 26% when packages are clustered into 4 configurations. Without pragmas, a single best configuration can give us a 7% improvement, and 9% with a 4-configuration clustering. These configuration changes have been shown to improve performance across architectures as well.

To achieve the maximum observed speedup, however, the entirety of GHC's inliner should be rethought. The data suggests that no single set of magic numbers will optimize all different types of programs. A newer inlining heuristic should also give more consideration to anonymous functions, and it should inline larger things. The complexity of this problem indicates that machine learning would be a good alternative to further hand tuning.

7 Related Work

There is much prior work in benchmarking, inlining, and program optimization, which we briefly summarize here.

Benchmarking. Prior work explores the creation of benchmarks that mimic real-world behaviour in JavaScript, as described by Richards et al. [2011b] and Mickens et al. [2010]. Blackburn et al. [2006] describe the selection of real-world open-source benchmarks for Java, and Pattabiraman and Zorn [2010] and Ricca and Tonella [2001] present systems for testing individual applications. Benchmark creation has been addressed via synthesis of benchmarks from real-world programs by Van Ertvelde and Eeckhout [2010], and more recently by Cummins et al. [2017] and Goens et al. [2019].

Inlining. The approaches closest to this work are by Cavazos and O'Boyle [2005] and Kulkarni et al. [2013] that target inlining in Java, exploring a parameter space of a simple decision tree. Later work by Sewe et al. [2011] assumed a fixed inlining heuristic and predicted likelihood of inlining to guide information propagation. More recently, Mosaner et al. [2021] investigate the use of machine learning to predict the impact on code size duplication. As Java is JIT compiled, the focus is on trading compilation time with execution time. For instance, Ochoa et al. [2021] aim to reduce compilation time at the expense of greater execution time. GHC, however, relies on aggressive, offline compilation where inlining is the key optimization. Rather than a simple decision tree, GHC has a complex inlining heuristic that users feel performs sub-optimally, requiring pragma guidance.

Search-based optimization. Iterative optimization finds performance improvement by trying multiple configurations of compiler optimizations and has been studied extensively [Agakov et al. 2006; Chen et al. 2010; Franke et al. 2005; Kulkarni et al. 2004]. More closely related work investigates parameters in compiler optimizations [Ansel et al. 2014; Ashouri et al. 2018; Chen et al. 2021; Monsifrot et al. 2002] and uses machine-learning-based optimization, as described by Wang and O'Boyle [2018]. Early works by Stephenson et al. [2003] explored the tuning of a compiler heuristic to drive low-level optimization, focusing on issues such as hyper-block formation which impact performance. Search of optimization spaces has also been used at higher levels of the compiler. Ganser et al. [2017] focus on parallelization and tiling within the polyhedra space, showing that a simple random search of configurations can deliver performance without the need for more expensive technologies such as genetic algorithms. Optimization search has been widely used in different compiler domains, as described recently by Leather and Cummins [2020]. All these approaches target imperative programs where the control-flow graph and function parameter types are statically known. Haskell is a higher-order polymorphic functional language where the treatment of functions, including inlining, dominates performance concerns.

Functional programming. The need to improve Haskell's performance is well known and, for example, documented by Jones et al. [2009] and Richards et al. [2011b]. Recent work by Shivkumar et al. [2021] addressed predictability of execution time, particularly for real-time programming relying on aggressive "defunctorization" which reduces code to first-order by the insertion of look-up data structures. While this increases runtime determinacy, it prevents performance improvement due to large inline function blocks.

8 Conclusions

It has been widely recognized that inlining is critical for GHC's performance; but its heuristic has not been updated for nearly 20 years, leading programmers to rely on pragmas. Inlining decisions are pervasive in the compiler source code and highly complex, relying on hard-coded "magic numbers" based on out-of date intuition. This complexity means that changing the heuristic or numeric values may lead to unexpected behavior and is a barrier to improvement.

In this paper, we explored the magic-number configuration space on modern benchmarks and showed that significant performance improvement is available, justifying an update of the inlining heuristic. However, the number of programs selected was small and focused on those with pre-existing inline pragmas. Future work will evaluate a broader range of benchmarks and platforms, and also investigate whether greater structural change to the inliner can further improve performance and maintainability.

Acknowledgments

We thank all reviewers for their feedback, Nick Smallbone for help with QuickCheck, and Artem Pelenitsyn and Simon Peyton Jones for help with GHC-related questions.

References

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. 2006. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization (CGO'06)*. 11 pp.–305. https://doi.org/10.1109/CGO.2006.37

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 303–316. https://doi.org/10.1145/2628071.2628092

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. 2018. A Survey on Compiler Autotuning using Machine Learning. *Comput. Surveys* 51 (09 2018), 96:1–. https://doi.org/10.1145/3197978

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

John Cavazos and Michael F. P. O'Boyle. 2005. Automatic Tuning of Inlining Heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*. IEEE Computer Society, USA, 14. https://doi.org/10.1109/SC.2005.14

Junjie Chen, Ningxin Xu, Peiqi Chen, and Hongyu Zhang. 2021. Efficient Compiler Autotuning via Bayesian Optimization. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1198–1209.

William Y. Chen, Pohua P. Chang, Thomas M. Conte, and Wen-mei W. Hwu. 1993. The Effect of Code Expanding Optimizations on Instruction Cache Design. *IEEE Trans. Computers* 42, 9 (1993), 1045–1057. https://doi.org/10.1109/12.241594

Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating Iterative Optimization across 1000 Datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. Association for Computing Machinery, New York, NY, USA, 448–459. https://doi.org/10.1145/1806596.1806647

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (Sept. 2000), 268–279. https://doi.org/10.1145/357766.351266

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. 2017. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 86–99. https://doi.org/10.1109/CGO.2017.7863731

Jack Davidson and Anne Holler. 1992. Subprogram Inlining: A Study of its Effects on Program Execution Time. *Software Engineering, IEEE Transactions on* 18 (03 1992), 89 – 102. https://doi.org/10.1109/32.121752

Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. 2005. Probabilistic Source-Level Optimisation of Embedded Programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Chicago, Illinois, USA) *(LCTES '05)*. Association for Computing Machinery, New York, NY, USA, 78–86. https://doi.org/10.1145/1065910.1065922

Stefan Ganser, Armin Grösslinger, Norbert Siegmund, Sven Apel, and Christian Lengauer. 2017. Iterative Schedule Optimization for Parallelization in the Polyhedron Model. *ACM Trans. Archit. Code Optim.* 14, 3, Article

23 (Aug. 2017), 26 pages. https://doi.org/10.1145/3109482

Andrés Goens, Alexander Brauckmann, Sebastian Ertel, Chris Cummins, Hugh Leather, and Jeronimo Castrillon. 2019. *A Case Study on Machine Learning for Synthesizing Benchmarks*. Association for Computing Machinery, New York, NY, USA, 38–46. https://doi.org/10.1145/3315508.3329976

Celeste Hollenbeck. 2022. ghc-benchmark-maker. https://github.com/CAHollenbeck/ghc-benchmark-maker.git

Don Jones, Simon Marlow, and Satnam Singh. 2009. Parallel Performance Tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/1596638.1596649

Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. 2004. Fast Searches for Effective Optimization Phase Sequences. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation* (Washington DC, USA) *(PLDI '04)*. Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/996841.996863

Sameer Kulkarni, John Cavazos, Christian Wimmer, and Douglas Simon. 2013. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 1–12. https://doi.org/10.1109/CGO.2013.6495004

Hugh Leather and Chris Cummins. 2020. Machine Learning in Compilers: Past, Present and Future. In *2020 Forum for Specification and Design Languages (FDL)*. 1–8. https://doi.org/10.1109/FDL50818.2020.9232934

Simon Marlow. 2005. *Haskell Cafe post by Simon Marlow*. https://haskell-cafe.haskell.narkive.com/jJTv7WgN/proposal-habench-a-haskell-benchmark-suite#post3 Accessed: 2021-04-06.

James Mickens, Jeremy Elson, and Jon Howell. 2010. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (San Jose, California) *(NSDI'10)*. USENIX Association, USA, 11.

Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2009. *Magic Number (Programming): Computer Programming, File Format, Software Documentation, Globally Unique Identifier, Enumerated Type, Hexspeak, NaN, Version ... OSCAR Protocol, AOL Instant Messenger*. Alpha Press.

Yaron Minsky. 2016. *A better inliner for OCaml, and why it matters*. https://blog.janestreet.com/flambda/

Antoine Monsifrot, François Bodin, and Quiniou René. 2002. A Machine Learning Approach to Automatic Production of Compiler Heuristics, Vol. 2443. 41–50. https://doi.org/10.1007/3-540-46148-5_5

Raphael Mosaner, David Leopoldseder, Lukas Stadler, and Hanspeter Mössenböck. 2021. Using Machine Learning to Predict the Code Size Impact of Duplication Heuristics in a Dynamic Compiler. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Münster, Germany) *(MPLR 2021)*. Association for Computing Machinery, New York, NY, USA, 127–135. https://doi.org/10.1145/3475738.3480943

Erick Ochoa, Cijie Xia, Karim Ali, Andrew Craik, and José Nelson Amaral. 2021. *U Can't Inline This!* IBM Corp., USA, 173–182.

Will Partain. 1993. The nofib Benchmark Suite of Haskell Programs. In *Functional Programming, Glasgow 1992*, John Launchbury and Patrick Sansom (Eds.). Springer London, London, 195–202.

Karthik Pattabiraman and Benjamin Zorn. 2010. DoDOM: Leveraging DOM Invariants for Web 2.0 Application Robustness Testing. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 191–200. https://doi.org/10.1109/ISSRE.2010.17

Simon Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming* 12 (July 2002), 393–434. https://www.microsoft.com/en-us/research/publication/secrets-of-the-glasgow-haskell-compiler-inliner/

Filippo Ricca and Paolo Tonella. 2001. Analysis and Testing of Web Applications. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada) *(ICSE '01)*. IEEE Computer Society, USA, 25–34.

Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011a. Automated Construction of JavaScript Benchmarks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 677–694. https://doi.org/10.1145/2048066.2048119

Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. 2011b. Automated Construction of JavaScript Benchmarks. *SIGPLAN Not.* 46, 10 (Oct. 2011), 677–694. https://doi.org/10.1145/2076021.2048119

Andreas Sewe, Jannik Jochem, and Mira Mezini. 2011. Next in Line, Please! Exploiting the Indirect Benefits of Inlining by Accurately Predicting Further Inlining. In *Proceedings of the Compilation of the Co-Located Workshops on DSM'11, TMC'11, AGERE! 2011, AOOPES'11, NEAT'11, & VMIL'11* (Portland, Oregon, USA) *(SPLASH '11 Workshops)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2095050.2095102

Bhargav Shivkumar, Jeffrey C. Murphy, and Lukasz Ziarek. 2021. Real-time MLton: A Standard ML runtime for real-time functional programs. *J. Funct. Program.* 31 (2021), e19. https://doi.org/10.1017/S0956796821000174

Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. 2003. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 77–90. https://doi.org/10.1145/781131.781141

Luk Van Ertvelde and Lieven Eeckhout. 2010. Benchmark synthesis for architecture and compiler exploration. In *IEEE International Symposium on Workload Characterization (IISWC'10)*. 1–11. https://doi.org/10.1109/IISWC.2010.5650208

Zheng Wang and Michael O'Boyle. 2018. Machine Learning in Compiler Optimization. *Proc. IEEE* PP (05 2018), 1–23. https://doi.org/10.1109/JPROC.2018.2817118