The Intel Labs Haskell Research Compiler

Hai Liu   Neal Glew   Leaf Petersen   Todd A. Anderson

Intel Labs

{todd.a.anderson,hai.liu,leaf.petersen}@intel.com   aglew@acm.org

Abstract

The Glasgow Haskell Compiler (GHC) is a well supported optimizing compiler for the Haskell programming language, along with its own extensions to the language and libraries. Haskell's lazy semantics imposes a runtime model which is in general difficult to implement efficiently. GHC achieves good performance across a wide variety of programs via aggressive optimization taking advantage of the lack of side effects, and by targeting a carefully tuned virtual machine. The Intel Labs Haskell Research Compiler uses GHC as a frontend, but provides a new whole-program optimizing backend by compiling the GHC intermediate representation to a relatively generic functional language compilation platform. We found that GHC's external Core language was relatively easy to use, but reusing GHC's libraries and achieving full compatibility were harder. For certain classes of programs, our platform provides substantial performance benefits over GHC alone, performing 2× faster than GHC with the LLVM backend on selected modern performance-oriented benchmarks; for other classes of programs, the benefits of GHC's tuned virtual machine continue to outweigh the benefits of more aggressive whole program optimization. Overall we achieve parity with GHC with the LLVM backend. In this paper, we describe our Haskell compiler stack, its implementation and optimization approach, and present benchmark results comparing it to GHC.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Compilers

Keywords Functional Language Compiler; Compiler Optimization; Haskell

1. Introduction

The Glasgow Haskell Compiler (GHC) is a robust optimizing compiler for the Haskell programming language, providing extensive libraries and numerous extensions on top of standard Haskell. GHC is widely used by the Haskell community as a standard development platform, and also serves as a vehicle for active programming language research. The Haskell programming language is widely used within the functional programming (FP) community, and has gained increasing traction outside of the FP world as well.

Haskell'13, September 23–24, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2383-3/13/09. . . $15.00.
http://dx.doi.org/10.1145/2503778.2503779

Despite the apparent complexity of the Haskell surface language, the design philosophy of the language is such that it is possible to reduce (or "de-sugar") the surface language to a surprisingly small core language. GHC in fact does this explicitly as part of its compilation strategy: most of the advanced features of the language are quickly eliminated, leaving only a relatively simple System F based intermediate representation (IR) known as Core [27, 29]. Much of the more advanced optimization technology in GHC is implemented as transformations on Core.

In addition to aggressive optimization, GHC also employs a highly-tuned virtual machine and garbage collector designed closely around the requirements of implementing the lazy semantics of Haskell efficiently. After optimization is performed on Core, programs are translated to an IR based on this virtual machine, the Spineless Tagless G-Machine (STG Machine) [25]. STG representations are then translated into *Cmm*, a variant of the C-- language [23] before passing to GHC's native code generator (NCG) or LLVM to generate binary executables.

This paper reports on an ongoing effort to compile Haskell by using GHC as a frontend to an existing functional language compiler built at Intel Labs that is largely language agnostic. We use GHC to perform de-sugaring and high-level optimization; intercept the Core IR from GHC and translate the lazy Core language into a strict, lower-level, general-purpose IR; perform aggressive whole-program compilation on this IR; and compile the result eventually to Pillar [2], also inspired by C--. Our choice of building a Haskell compiler by marrying two compiler platforms together is deliberate. Being able to support the Haskell language and various GHC extensions gives us instant access to a large set of real world libraries and programs, as well as the opportunity to contrast and compare our methodologies in compiling functional languages with those taken by GHC.

We observe that the promised simplicity of the Core IR has by and large been borne out, but that the consequent complexity of the interactions with the runtime system makes using GHC in this fashion more difficult than it might at first seem. We present results across a wide range of benchmarks and show that for some programs we are able to add substantial value over the core GHC system, thereby demonstrating that our more aggressive optimization can overcome the lack of a specialized runtime; but that for other programs we are still a long way from being able to match the performance of the tuned virtual machine and runtime. Overall we achieve parity with GHC with the LLVM backend, and achieve a 2× speedup on a set of modern performance-oriented benchmarks. We refer to our compiler as the Intel Labs Haskell Research Compiler (HRC).

We make the following contributions:

- An experiment to reuse GHC as a Haskell frontend, connecting it to our own middle-end and backend, demonstrating that GHC's external Core is indeed easy to use, but reusing GHC's libraries and achieving full compatibility are harder.

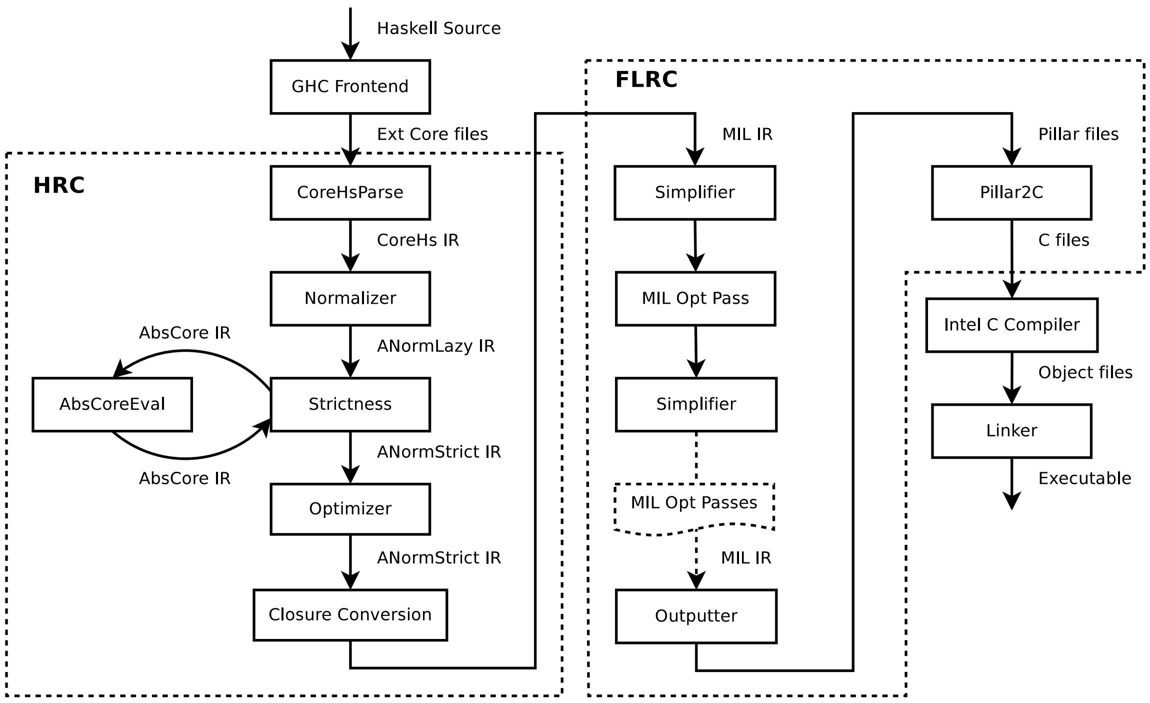

Figure 1. IRs and Passes of Intel Labs Haskell Research Compiler Pipeline

- A detailed description of an alternative approach to implementing Haskell based on a traditional explicit-thunks IR and traditional compiler optimizations tailored to exploit functional-language properties.
- A novel design of a functional compiler IR that combines low-level control flow with a high-level object model, thereby enabling a number of optimizations not yet available to other compilers, including optimizations on immutable arrays with initializing writes. To the best of our knowledge, this concept of initializing writes is novel to our compiler.
- Evidence confirming the highly-tuned nature of GHC's STG Machine and garbage collector, but also showing that GHC leaves performance on the table for certain applications.

2. FLRC

HRC is constructed as a frontend to a more general functional language compilation platform called the Intel Labs Functional Language Research Compiler (FLRC). We begin by giving an overview of the principle components of our compiler. We discuss the approach taken to compilation, give a general description of the main intermediate representations used, and describe at a high level the important optimizations performed.

2.1 Architecture

Figure 1 gives the overall pictorial view of our compiler pipeline. The boxes represent transformation passes of the compiler, and lines represent data flowing through the compiler, annotated with the specific IR going into and coming out of each pass. When we compile a set of input Haskell files, we first invoke GHC to compile them to external Core files, read them back in, then go through various internal transformations and multiple IRs, before outputting a program in a language called *Pillar* [2] (an extension to the C language inspired by C-- [23]). A tool called *Pillar2c* is used to translate the Pillar code to C code, and the Intel C compiler is used to produce the final machine executable.

At the core of FLRC is a language agnostic intermediate representation called *MIL*, with an associated set of optimization passes. The majority of the optimization in the compiler takes place at the MIL level. From MIL code, we generate low-level Pillar code. Pillar provides facilities to support garbage collection and other essential functionality for supporting high-level languages, but is otherwise essentially C (and is in fact implemented as an extension to C).

Specializations of FLRC to concrete languages are realized by implementing language specific frontends targeting MIL as a backend. FLRC was originally developed to support an experimental strict functional language [11], which has a separate frontend than HRC. Both frontends implement a set of language specific optimizations on higher-level intermediate representations, followed by a globalization/closure-conversion pass which lowers these high-level representations into MIL.

2.2 MIL

MIL is a loosely typed, control-flow-graph (CFG) based, closure-converted, intermediate representation. The essential design philosophy behind MIL is to maintain a relatively high-level object representation while using a low-level CFG representation of program control flow. Our observation is that there are dramatic optimization benefits that can be obtained from leveraging the functional language properties of *immutability* and *memory-safety*. In order to take advantage of these properties, our intermediate representation uses an object-based model of memory. We do not view memory as a large array of bytes which may be arbitrarily mutated, but rather as a collection of objects with well-defined initialization, reading, and (for mutable objects) updating operations. On the other hand, while maintaining high-level object representations provides great optimization benefits, we argue that there are significant benefits to a low-level representation of control flow with very few downsides.

Since we are particularly interested in optimizing code which operates on high-level aggregate objects such as immutable arrays, it is important that the initialization code (including the control-flow) for these objects be expressible directly in the local CFG at their allocation point. In order to combine high-level objects with a low-level control-flow representation, we use *initializing writes*—a write to an object field that is guaranteed to be dynamically the only write to that field. Initializing writes allow us to break down the initialization of large (even statically unbounded) objects into sequences of writes or loops while preserving immutability information. From the perspective of optimization, initializing writes to fields can be optimized in essentially the same way as an aggregate immutable object construction. For example, a read from a field can be freely replaced by the operand of an initializing write to the same field. In order for this perspective to hold, initializing writes must satisfy two unchecked invariants: a field cannot be read until it is initialized and a field cannot be initialized twice. To the best of our knowledge, the use of initializing writes is novel to our compiler. A more standard approach used in compilers such as MLton [30] and GHC is to allocate and initialize small heap values atomically, but to initialize large aggregates using mutable operations and only subsequently coerce the result to an immutable representation.

2.2.1 Types

MIL is loosely typed in that every variable has a type, and those types must satisfy certain properties. However, it is not *type safe* since the types of heap values are not accurately tracked, and consequently the correctness of heap accesses cannot be statically checked. Types are used in MIL primarily as means of tracking and maintaining garbage collection information and secondarily as an engineering methodology to improve the correctness of the compiler.

In addition to the usual constraints of correctness with respect to program semantics, it is essential that the final generated code maintain a GC-safety property. Accurate garbage collection imposes certain requirements on programs, needing information about which variables and object fields contain GC-managed references. Aggressive optimization such as inter-procedural object unboxing may change the GC status of variables and fields of objects [10, 19]. The MIL language must support the tracking of such information, and the MIL optimizations must successfully maintain this information. In the IR design, we represent this information using meta-data associated with object allocation sites and using types on variables.

2.2.2 Effects

While MIL optimization focuses primarily on purely functional (non-side-effecting) code, side-effecting code must nonetheless be safely handled by the compiler. Side-effects arise both from source code (e.g. monadic IO in Haskell) and because of lowering higher-level language constructs to a lower-level language (e.g. setting and reading exception handler data-structures). In order to preserve correctness without forcing the compiler to make overly conservative assumptions, MIL makes use of effect annotations that mark functions and thunks with a set of side-effects that may be unleashed by calling or evaluating the function or thunk. Effect annotations consist of subsets drawn from the full set of effects including potential non-termination, heap reads and/or writes, generative allocation, input/output effects, exception throwing, and others.

2.2.3 IR structure

A MIL program is a set of global objects named by variables, one of which (which must be a code function) is a designated entry point for the program. A symbol table maps variables to various useful meta-data, most notably the type. Objects in MIL are either code functions (corresponding approximately to C functions), primitive values (such as integers or floating pointer numbers), or abstract heap values. Heap values are allocated, initialized, and manipulated via instructions (making extensive use of initializing writes to preserve immutability properties). Inter-procedural control-flow is effected via call and eval instructions. Calling conventions provide for either calls/evals via abstractly represented closures or direct calls/evals to code functions in the case that the target code function is known and no closure is required. Annotations on call sites allow (conservative approximations of) control-flow information either apparent in the initial program or computed via control flow-analysis to be recorded directly in the intermediate representation as sets of code pointers potentially reaching the call site, providing some of the benefits of de-functionalization [26].

To represent instructions, MIL uses a variant of static single-assignment (SSA) form similar to that used in compilers such as the MLton compiler [30]. Basic blocks are parameterized over input variables (playing the role of phi functions from standard SSA form), contain a sequence of instructions, and are terminated by either an inter-procedural or intra-procedural control transfer. Blocks contain zero or more successors, each explicitly given as a target of the control transfer. Non-local control flow such as exceptions can be implemented via a second-class continuation mechanism, and such exception edges are made explicit in the intermediate representation using annotations on transfers and call/eval sites. Input variables for blocks are defined in the control transfer targeting the block. This style of SSA fairly closely resembles (and provides many of the benefits of) the use of continuations in a continuation-passing style compiler, but with the added benefit of segregating the local control-flow from the (potentially) truly inter-procedural control-flow.

2.3 MIL optimization

FLRC is de facto a whole program compiler in that whole program compilation is the only supported compilation mode. Whole program compilation allows us to take advantage of the significant optimization opportunities presented by having all of the program code present for analysis. However, there is nothing in our compilation strategy that requires whole program compilation. In particular we have chosen not to do whole program de-functionalization [26], and our global optimizations, while clearly benefiting from access to the entire program text, do not assume so.

The primary goal of the MIL optimizer is to leverage immutability and memory safety properties of source languages to provide aggressive optimizations that cannot feasibly be performed on arbitrary mutable code. The high-level object model provides significant benefit to the optimizer, since the various objects encode strong invariants about when and how the object can be mutated, and what classes of objects might dynamically reach a given instruction. At the same time, the low-level control flow structure allows for high-level idioms such as loops expressed through nests of mutually recursive functions to be expressed as local control flow. In principle, optimizations that apply to loops expressed as control-flow graphs apply equally well to loops expressed via functions. This is true in the same sense that any intra-procedural optimization can be done inter-procedurally: that is, it is possible to do so, but requires substantially more effort, since each such optimization must rediscover and reconstruct the portion of the inter-procedural call graph which corresponds to local control-flow (that is, does not escape, and is well-behaved in other ways), and hence to which the intra-procedural optimization applies. Performing intra-procedural optimizations inter-procedurally can also complicate the cost-model against which the optimization must be designed significantly: for example, the inter-procedural analogy to loop-invariant code motion in general requires introducing a wrapper function to serve as a pre-header, the cost of which may outweigh the benefit of the code motion. Our experience leads us to believe that it is vastly simpler and more efficient to just discover once and for all which portions of the inter-procedural call graph implement local control-flow, and to represent these portions directly as local control-flow to which standard intra-procedural optimizations (suitably adapted) can be applied.

2.3.1 Optimizations

Given the emphasis on using standard intra-procedural optimizations where possible, a key element of MIL optimization is turning inter-procedural control-flow into local control-flow. In the MIL this is done by two sets of optimizations. The first is a contification [9] pass which turns uses of (mutual) recursion into loops. This may be thought of as a very generalized version of the standard approach to turning self tail-recursive functions into loops (which is also done as part of this pass). This pass is very effective at eliminating inter-procedural control-flow. Secondly, in addition to contification, several different inlining passes are run using different heuristics. In one frequently run pass, functions known to be small are inlined aggressively. Another pass performs more aggressive inlining using fairly standard cost-budget inlining heuristics. A final inlining pass uses a static profile estimation approach based on work by Wu and Larus [31] to perform selective inlining at (estimated) high-frequency call sites.

Another large set of mostly intra-procedural optimizations are performed by a simplifier in the general style of Appel and Jim [3]. In order to implement this efficiently, before each run of the simplifier the MIL intermediate representation is wrapped in an imperative data structure notionally implementing the linear time approach described by Appel and Jim, and previously implemented by Benton et al [4]. A general worklist algorithm is run performing a large set of dataflow based optimizations including dead-code elimination, constant and copy propagation, and other general simplifying reductions. The simplifier is designed to avoid increasing program size and to only perform optimizations which strictly improve the program: consequently it can safely be run with extremely high-frequency (before and after every other optimization). The imperative representation has proved extremely efficient in practice, and each run of the simplifier contributes only negligibly to overall compile time.

The compiler also implements a number of inter-procedural representation optimizations using a field-sensitive, unification based flow analysis [10, 19]. The main analysis can be roughly thought of as computing a set of equivalence classes on variables and object fields such that any two members of different equivalence classes can be guaranteed never to contain the same dynamic heap value. Given such an analysis, the compiler can use the information computed by it to perform a number of representation optimizations in a GC safe manner [10, 19]. For example, small, single-field, immutable objects (such as boxed floating point numbers) are replaced inter-procedurally by the contents of the object. This is done even when the object in question is placed into the heap as part of other mutable or immutable objects, and is always done *without* introducing any additional allocation or projection operations, even at escape points. Other optimizations performed using this analysis include inter-procedural dead-code and dead-field elimination, function argument flattening, inter-procedural constant propagation, and control flow analysis. The analysis is also used to eliminate the overhead of checking the evaluation status of thunks where possible. The choice to use a unification based algorithm for this analysis was driven in part by efficiency concerns, and in part by the need to deal with GC safety issues. The analysis has proved very scalable in practice, and generally very effective. The limitations of unification based analysis do at times become apparent however, and extending this with a subset based analysis remains an area of interest for future work.

A number of intra-procedural optimizations target loops in the control-flow graph. A loop inversion pass is used to attempt to rewrite inner loops into a more amenable form for optimization. Specifically, it turns top-test loops into bottom-test loops in which once entered, the loop is guaranteed to run its body at least once. This transformation allows loop-invariant code to be moved out of loops in a non-speculative fashion, avoiding performance and correctness issues associated with speculative optimization. A loop-invariant code motion pass does such non-speculative movement, when safe to do so. Finally, a SIMD vectorization pass is performed to attempt to create SIMD vector versions of inner loops [21]. A key advantage of performing vectorization in MIL is that the dependence analysis problem for immutable arrays is vastly more tractable than the generalized problem for mutable arrays. Our vectorizer is able to vectorize loops for which the underlying C compiler is not able to safely produce vector code.

A number of supporting optimizations such as common sub-expression elimination, effect analysis, code function escape analysis, recursive function analysis, control-flow graph simplification and redundant branch elimination are also implemented and are run either as standalone passes or as sub-components of other passes.

Our initial implementation focus with the MIL optimizer, while supporting lazy code via thunks, was intended to target code in which thunking was relatively rare. From the perspective of the optimizer, thunks are pernicious not just because of the overhead they incur directly, but more generally because they dramatically obscure the control-flow of a program and hence greatly reduce the effectiveness of the optimizer. It is important to note that this is completely unrelated to the choice of using a strict-by-default intermediate representation: whether laziness is explicit or implicitly represented, the program semantics remain the same and the optimizer must perform the same reasoning about control flow. Since thunks are vastly more common in Haskell, we have begun implementing optimizations targeting thunked code specifically. One current optimization pass attempts to recognize and mark thunks which are either already values, or which can safely be evaluated at their definition site and passed as values. A second optimization makes a preliminary attempt at performing inlining of thunks by using a data-flow analysis to discover thunks that are evaluated on every path from their definition and to evaluate these strictly. This work is preliminary, but has shown good results so far. For many classes of Haskell programs, we see significant remaining opportunities.

2.4 Pillar

FLRC and Pillar [2] were concurrently developed in the same lab and one goal of the FLRC project was to act as a test case for language development on top of Pillar. Pillar is a language, compiler, and runtime that provides programming language infrastructure. The idea behind Pillar is to allow language developers to focus on compiler optimizations unique to that language and on runtime code for unique aspects of those languages. The Pillar infrastructure optimizes and provides support for features that are common to many languages and runtimes.

The core idea of Pillar is quite similar to (and inspired by) the C-- language [23], with which it shares many common concepts. A key difference between Pillar and C-- is that Pillar is implemented as an extension to C, rather than as an entirely separate language. This allows for the reuse of the numerous existing tools available for compiling, debugging, and performance-tuning C code. In addition, this approach makes it easy to incorporate existing C code into Pillar programs, since most C code can simply be compiled as Pillar code with no modification.

GHC		HRC	
Desugaring, type analysis, Core-to-Core transformation		Same process, since it uses GHC as frontend	
STG	Functional language, object based memory model, and optimized for currying and thunks	MIL	SSA, CFG based blocks with explicit transfer, object based memory model, but conventional
Cmm	Based on C−−, CFG based blocks, low-level types, and custom calling convention	Pillar	Inspired by C−−, C types, C calling convention
LLVM or NCG	Portable LLVM bitcode, or direct assembly generation	Intel C/C++ Compiler	Portable C code compiled to assembly
Runtime and GC optimized for currying and thunks		Conventional runtime and GC	

Table 1. Comparison between GHC and HRC

The Pillar language extends C with a small number of additional constructs including parallelism, a *Ref* type identifying GC-managed pointers, second-class continuations, tailcalls, and calling conventions for the integration of Pillar and ordinary C code. The Pillar compiler infrastructure is responsible for taking Pillar code, lowering it to machine code, and in conjunction with the Pillar runtime, providing support for stack walking, root-set enumeration (RSE), tailcalls, composable continuations, and transitions between managed and unmanaged code.

Originally, the Pillar compiler was implemented as modifications to the Intel C/C++ Compiler. This approach allowed *Refs* and tailcalls to be implemented with no runtime overhead but also required frequent reintegration of our modifications with an ever changing compiler codebase, which quickly became burdensome. Therefore, we experimented with a different Pillar implementation that translates Pillar to C using the *Pillar2C* translator and then uses an unmodified (and up-to-date) Intel C/C++ compiler to compile the translated output to binary. Pillar2C uses a shadow-stack approach to support *Refs* and implements a number of optimizations for the shadow-stack and tailcalls. With these optimizations, the average Pillar2C runtime overhead when compared to the native compiler was approximately 10%.

The runtime for our compiler uses a modified version of the TGC garbage collector [1] that was created for the first FLRC frontend. These modifications include the addition of Haskell-specific features such as weak pointer objects and the ability to perform thunk indirection removal. In the original frontend, writes to global objects were minimal due to an eager evaluation strategy that worked well with TGC's private nurseries. Conversely, as described by Marlow and Peyton Jones [16] and verified by us, the lazy evaluation strategy of Haskell produces many more writes to global objects. These writes cause very frequent private nursery collections in TGC and the overhead from these collections can increase the runtime of an application by several times. These collections could be minimized through the use of a read barrier integrated with thunk evaluation [16]. However, we have not implemented this approach since we felt it broke Pillar modularization. Instead, we use a non-generational mark-sweep-compact mode in TGC without private nurseries. This illustrates an advantage of GHC's integrated runtime.

3. HRC

GHC compiles Haskell source programs to a typed internal representation called Core that is very close to System F [27], and is able to export an external representation of the Core program with well defined syntax [29]. HRC uses GHC as a frontend to compile from Haskell source to Core, and then takes GHC's external Core and translates it to MIL, before passing down to the rest of FLRC compilation pipeline.

Table 1 summarizes the difference between GHC and HRC at different compilation stages. Most notably, GHC's STG representation is drastically different from the MIL IR employed by HRC/FLRC. The former is still of a functional style with lambda abstraction and application, while the latter follows a SSA style with CFG based block structure; the former has a custom design to handle fast currying [15], while the latter stays within a conventional heap object model.

The CFG based block structure used by MIL is similar to low-level control flows found in Cmm or LLVM, but it's the high-level object representation that puts MIL in a unique position to exploit properties of functional programs. In contrast, much of the high-level type and meta information is lost once GHC lowers a program from STG to Cmm (and to that extent, LLVM bitcode). Sophisticated analysis is required to even attempt to derive the invariant that could be encoded into the meta-data (e.g., which object fields are written to only once) from IRs that use a lower-level memory model such as Cmm or LLVM bitcode.

As a consequence of the MIL design, we choose to intercept the intermediate Core representation of GHC rather than STG or Cmm because we want to keep available the rich type information in Core to help build object type and meta-data in MIL. However, the task of connecting GHC as a frontend to FLRC is not as simple as a mere translation from GHC Core to MIL. There are numerous practical challenges involved in making this work out:

- GHC is an incremental compiler, which compiles each module in relative isolation (modulo extensive cross-module inlining), while FLRC is currently a whole program compiler.
- The intermediate representation of GHC is essentially a lazy functional language based on System F, while MIL is a CFG-based strict language with first-order functions. GHC annotates impure operations in its IR via state-passing, whereas MIL uses an explicit effect annotation system.
- GHC compiled programs rely on (and are sometimes tightly coupled with) the GHC runtime, a complex and highly tuned system, for implementing critical features including GHC primitives, multi-threading, garbage collection, etc.

In the remainder of this section, we describe these differences in more detail and discuss the impedance matching required in order to integrate the two compilers. We begin by discussing the modifications required to GHC itself in order to enable its integration into our compiler pipeline.

3.1 Modifications to GHC

Outputting external Core GHC has both an internal Core and an external Core representation [29], with the latter intended to support the exchange of programs with the outside world. Unfortunately over the years this part of GHC has not been fully maintained as it is not widely used. In order to make use of this facility, we have brought this code back into a sufficiently usable state to cover the large fragment of Core that we require for correctness, as well as some additional annotation information such as strictness information that are important for performance reasons.

A related issue for our purposes is that in order to impedance match between the GHC incremental compilation model and the FLRC whole-program model, we require the ability to access not just the external Core representation of the main program, but also the installed libraries. To deal with this, we have modified the build process of GHC and the related Cabal library tool to output external Core files when compiling libraries, and to copy Core files along with standard binary files when installing libraries.

Library Linking Because of differences in the runtime model, we cannot directly link HRC compiled object files with GHC's runtime or with GHC compiled libraries. However, many programs and libraries contain C or FFI code fragments that are not representable in external Core, and that will result in link errors if not handled properly. Besides, recent GHC will also automatically produce stub codes (in C) when compiling certain form of FFI imports, which may then be referenced in the generated external Core. To solve these linking problems, we add a new *-fstub-only* option to tell GHC to produce object files that contain only foreign code segments. If we use this modified GHC to compile a Haskell library, we will get a binary library file containing objects with only foreign definitions in them. When HRC takes the external Core as input, it is able to find foreign function definitions in these library files at link time. The Cabal library has also been modified to use this option when compiling and installing Haskell libraries for HRC.

Arbitrary precision integer FLRC has internal support for arbitrary precision integer as a primitive type, while GHC provides it through one of the two libraries: *integer-simple* or *integer-gmp*. The former is pure Haskell and portable, but it is not a high performance library. The latter links with GMP C library, but contains `C--` code as well as GC hooks that are tied into GHC's runtime, and therefore does not properly work with FLRC. Our solution is to modify GHC to declare a set of primitives that operate on integers, but leave them as unimplemented. Then we modify the integer-simple library to implement its API in terms of these newly added primitives so that we can eventually intercept them in our compiler and map to FLRC's built-in integer primitives.

Building GHC Building GHC is a rather complex job that involves multi-stage compilation in which a stage-1 compiler is used to build a stage-2 compiler, and so on. It is critical that our modifications do not break GHC's own functionality during the build process. However, some of our modifications such as the changes to the integer-simple library pose a challenge: we either have to fully implement the new integer primitives in GHC itself, or risk having broken libraries that prevent the next stage from building. Our (not entirely satisfactory) solution is to hack the compilation process to have GHC compile both the modified and unmodified versions of integer-simple, install binaries from unmodified version to support continuing building GHC, and install external cores and stub-only libraries from the modified version to support HRC. Care must be taken to make sure both versions export exactly the same set of names, and mismatching internal names do not accidentally leak into header files as they are randomly generated by GHC. Mismatches in function names will either produce errors of undefined symbols, or lead HRC to retrieve a wrong definition from the Core files, which is even more hiderous. There are also a number of other modifications to the base library that require this sort of handling due to differences in runtime support.

Immutable Arrays MIL is based around immutable arrays with initializing writes, whereas the GHC array and vector libraries tend to create mutable arrays, initialize them with writes, and then *freeze* the mutable array to an immutable array type. This freeze operation just returns its input but with a different type. Although MIL is capable of handling mutable arrays and repeating writes, our optimizations are all targeted at immutable arrays and initializing writes. Therefore, we have modified GHC's `vector` library to target our immutable arrays. These modifications include adding new primitive types in GHC for our immutable arrays (GHC has its own immutable arrays, but we decided to keep those separate), adding new primitive operations for creating without initialization, initializing writes, length, and reading of these new immutable arrays, and modifying the `vector` library itself to use these new primitive types and operations.

3.2 Architecture of HRC

The overall architecture of HRC is shown in Figure 1. Compilation begins by first invoking the modified GHC executable to compile the input program to external core, which is then read back into HRC and parsed into an internal representation of Core called *CoreHs*. A dependence analysis is performed on this representation to determine what other Haskell modules are required to complete the program. Since our modified GHC has already compiled and installed GHC libraries along with their external Core files, they can then be read into HRC based on the results of the dependence analysis. This process proceeds transitively until the full Haskell program is read in. This process also serves to determine any required linking options for external libraries.

The result of this process is a representation of the entire Haskell program that is to be compiled, in a representation fairly similar to that used by GHC itself. After some initial cleanup work, this program is passed through two additional representations before being translated to MIL, the main optimization IR discussed in Section 2.2.

3.3 Lazy A-Normal Form

The first transformation in the HRC frontend translates the CoreHs code into a lazy A-Normal form [8] language called *ANormLazy*. Some cleanup work such as primitive and constructor saturation is performed as part of this translation. The primary transformation performed on this intermediate representation is a strictness analysis pass.

Strictness Analysis The purpose of a strictness analyzer is to annotate variable bindings with strictness information. A function f is strict in its argument if and only if $f \perp = \perp$. Instead of calculating on the actual value domain, an abstract interpretation of f operates on an abstract domain of two or multiple points. However, due to the limitation of modular compilation, GHC only keeps limited information of the strictness of each function in the interface (`.hi`) file, and therefore sacrifices accuracy in exchange for efficiency and modularity.

Since our compiler functions as a whole program compiler, we suspected that there might be an opportunity to uncover strictness properties of the source program that might have been missed during modular compilation. As an experiment, we have implemented a strictness analysis pass for the ANormLazy IR. The analysis takes an abstract interpretation approach over an abstract representation derived from ANormLazy called *AbsCore*. Our initial implementation uses a relatively simple algorithm described by Peyton Jones and Partain [22], but we hope at some point to replace it with a more complex algorithm such as that used by Jensen et al [12] to better handle higher-order functions. This optimization has proved less effective than we had hoped, but we continue to feel that there are opportunities to be had in this domain, in part because of our experiences with simple ad hoc strictness analyses performed in later phases of the compiler.

3.3.1 Strict A-Normal Form

The final and most significant intermediate form change in the HRC frontend is the translation from the ANormLazy language in which laziness is implicit, to a strict A-Normal form language called *ANormStrict* in which laziness is represented explicitly. The ANormStrict IR provides primitive thunks for suspending the computation of terms and explicit *eval* operations for forcing a thunk and memoizing its result. This is usually regarded as a more conventional treatment of handling laziness, as compared to the optimized design in GHC's STG machine [25]. Bolingbroke and Peyton Jones have also proposed a strict Core for GHC [6], advocating the benefits of representing laziness explicitly.

Because variable bindings in ANormLazy are already annotated with strictness information, translating from ANormLazy to ANormStrict is a relatively straight-forward process. For each strict expression binding in the lazy language, the strict code must evaluate the expression to a value, bind a fresh variable to the result, and wrap the fresh variable in a thunk bound to the original variable. For each lazy binding in the lazy language, the strict code simply constructs a thunk containing the translated expression and binds the original variable to it. Unboxed primitive bindings are simply translated directly to strict bindings. Case constructs which force the computation of thunks are translated into uses of the ANormStrict primitive eval construct which computes, memoizes, and returns the results.

We choose in this approach to wrap all boxed values explicitly in thunks, even when they are values. An alternative approach is to allow values to be subsumed into the class of thunks: that is, allowing strict bindings to simply bind the value to the original variable. In this case, the eval primitive must be prepared to dynamically distinguish between values and thunks. By choosing to wrap all boxed values explicitly, we allow our backend to choose whether to represent indirections explicitly, or to simply treat them as static coercions to the thunk type (relying on the runtime to distinguish between values and indirections). Our runtime can be configured to treat indirections in either manner simply by passing a flag to the compiler.

3.3.2 ANormStrict optimizations

While the main body of optimization is performed after translation to MIL code, it has proved very beneficial to implement a small set of cleanup and language specific optimizations in the frontend. The first reason for this is that the translations through the various frontend intermediate forms can be made much simpler if they are not required to produce perfectly “clean” code. It is often convenient to permit variable-to-variable moves to be introduced, or to use wrapper functions to ensure primitive saturation, etc. This kind of convenience code is easily eliminable, but interferes with the effectiveness of the closure converter if not actually eliminated.

The second reason for performing optimizations at the ANormStrict level is that certain language specific optimizations are simpler and more effective when performed at that level, both because of the more structured nature of the intermediate representation, and because of additional language specific invariants of Haskell programs. For example, strictness properties are much simpler to compute at the ANormStrict level, in part because the Haskell exception semantics [24] allows more code motion than is available after translation to MIL.

There are three main groups of optimizations performed on the ANormStrict language: general shrinking simplifications, uncurrying, and strictness. Shrinking simplifications are implemented using a fairly standard down and up traversal of the intermediate representation, performing dead code elimination, shrinking reductions, copy and constant propagation, thunk specific optimizations, and various other minor code improvements.

The uncurrying optimization is a simple syntactic optimization which rewrites curried functions as wrappers around uncurried functions, and replaces all saturated known applications of each curried wrapper by a call to its uncurried version. This approach was easy to implement and gave good improvements in runtime. However, the overhead of curried functions continue to be an issue in some benchmarks, suggesting that more sophisticated control-flow analysis based techniques [5] might be beneficial (either at the ANormStrict level, or in MIL). We have also considered experimenting with dynamic techniques such as those described by Marlow and Peyton Jones [15], but would prefer to explore static options first, since dynamic options impose an overhead even when they are not used.

The strictness optimization is a very simple but surprisingly effective dataflow based approach that attempts to find (interprocedurally) for each thunk variable the earliest program point at which it is guaranteed to be evaluated along all subsequent paths. If a thunk is guaranteed to be evaluated on all paths from its definition, then it can be evaluated eagerly, the thunk statically replaced with an indirection, and all syntactically visible uses replaced with the underlying computed value (the *unboxed* version). Otherwise, the thunk can be evaluated at the earliest point at which it is guaranteed to be evaluated and all subsequent syntactic uses (including arguments to known functions) replaced with the unboxed version. Some care must be taken to avoid incorrectly permuting computations that exhibit control effects (such as non-termination) with effectful code, since the GHC state passing representation does not sequence control effects with input/output effects.

The analysis for the strictness optimization traverses the program in a down and up fashion, performing a recursive top down analysis and then summarizing the results of the analysis on the returns from the recursive calls. Each function body is analyzed as it is reached to produce a procedure summary indicating in which arguments and free variables it is strict. Procedure summaries are used to incorporate strictness information on function arguments and free variables when calls to known functions are encountered. Summaries for recursive (or mutually recursive) functions are computed by iteratively re-analyzing until a fixed point is reached. The strictness optimization also performs dead code, dead argument, and dead field elimination simultaneously with strictness since the analysis required is essentially identical.

These optimizations were easy and quick to implement and have proved effective in eliminating a fair bit of the obvious cruft and low-hanging fruit. There remain substantial opportunities for further optimization at this level.

3.3.3 Closure conversion

One of the primary requirements for the translation from the ANormStrict language to MIL is the representation of functions (and thunks) as closures. In principle this is straightforward: a valid implementation is simply to compute the set of free variables of every function and place those in its closure. However, substantial benefit can be obtained by refining this in a number of ways.

Firstly, closure size can be reduced substantially by choosing to represent globally available small objects as static globals, which do not need to appear in closures. We refer to this process as *globalization*. While it is possible to perform globalization independently from closure conversion, the result is less effective than performing the analysis simultaneously with the closure analysis. The reason for this is that a closure can only be represented by a global if all of its free variables are globals, which in turn may depend on the choice of which closures are represented as globals. Proper globalization then is mutually dependent on closure conversion, and consequently we perform both analyses simultaneously.

Secondly, many closures for non-escaping functions can be eliminated entirely in the case that all free variables for the function are available (in the sense of either being directly in scope or in scope via an enclosing closure, or being global) at all call sites. Such functions can avoid having closures allocated at all, instead taking their arguments directly as additional parameters at each call site (a so-called *flat call*). Since flat-called functions do not require a closure, the choice of which functions to flat-call is again mutually dependent on globalization. This formulation of flat-calling is safe for space, since it never adds free variables to other closures in which they were not already present, and does not increase the live-range of variables.

Finally, call graph control-flow information that is apparent in the pre-closure converted program becomes obfuscated in the post-closure converted program if some effort is not made to preserve it during closure conversion. The MIL representation supports the annotation of call sites with the set of code pointers which may (conservatively) reach the call site: this is sufficient to preserve the pre-closure conversion control-flow information. It is straightforward to make the closure conversion algorithm preserve information matching up function and thunk variables projected from closures to the original function or thunk definition to which they correspond, and hence to use this information to build the initial call-graph annotations in MIL. While subsequent control-flow analysis may improve these annotations further, choosing not to lose the already present control-flow information proves very beneficial in bootstrapping the process.

Implementing this small set of extensions to a basic closure conversion algorithm provided significant improvements in the performance of the generated code for relatively small implementation effort. More sophisticated control-flow analysis based approaches have been considered, but we have not yet had the resources to experiment with this. One key limitation of our approach is that we do not globalize thunks except in the case that they are statically known to be values. In general, thunk globalization is not safe for space, since a thunk might compute and memoize an arbitrarily large object which as a global would remain live for the duration of the program. GHC solves this problem elegantly by using the garbage collector to decide dynamically which globals to enumerate and hence permitting objects computed by global thunks to be garbage collected [14]. We do not currently support this, and consequently we avoid globalizing all computed thunks.

3.4 Limitations/unimplemented

While our goal is to support as much of the GHC functionality as possible (including GHC extensions to Haskell), there are several known deficiencies in this regard (and of course, possibly unknown ones as well). The notable limitations that we are aware of are as follows:

1. We do not currently implement the correct semantics for propagating exceptions through thunks. Re-evaluating a thunk which exited with an exception will produce an error instead of re-raising the exception. Addressing this could have some adverse effect on the performance of thunk intensive code, but is largely irrelevant to the class of benchmarks on which we have focused, which have little or no laziness in performance critical sections.
2. Asynchronous exceptions are not supported. We do not currently see any path to addressing this limitation given our language agnostic runtime representation.
3. Although we have implemented many GHC primitives related to multi-threading and concurrency, we do not support lightweight threads, or GHC *sparks*, partly because of the complexity involved in designing their schedulers. We choose to map each `forkIO` invocation to creating a new thread using third party libraries such as pthread (POSIX thread).
4. There are still some known quirks related to the foreign-function interface and linking. In some infrequent cases, GHC decides to inline a foreign call, preventing us from correctly computing the external library to which the code must be linked.

4. Performance

We measure the performance of HRC using a set of benchmarks from a number of sources. The majority of them were taken from the *nofib* benchmark suite [18] which was designed to compare the performance of different Haskell systems. We tried to select a balanced set of nofib benchmarks, both lazy and strict, ranging from list manipulation, to big number arithmetic, to array computations. Many of the nofib benchmarks were written more than 20 years ago, and they often do not make use of modern GHC libraries such as `Data.Vector`, but we still feel that they are representative of typical Haskell programs, especially when we consider their runtime behaviors.

Besides nofib programs, we have also added a number of modern performance oriented Haskell programs mostly taken from the graphics, scientific computing, and finance spaces. These benchmarks have been our primary focus in tuning the optimizations in our compiler. Many of these benchmarks spend much of their time in array computations utilizing either the `vector` or `repa` libraries [13]. These benchmarks are often relatively strict in nature, either explicitly through programmer annotation or implicitly via compiler optimization. For those benchmarks written by us, we have generally tried our best to maintain an idiomatic functional style rather than littering the programs with lower-level imperative code.

All benchmark tests were conducted on a 2.7GHz Intel Xeon E5-4650 machine running Windows Server 2008. All benchmarks were compiled to 32-bit binaries using a standard GHC 7.6.1, GHC 7.6.1 with LLVM 2.9 backend, and HRC with our modified GHC 7.6.1 frontend and Intel C/C++ Compiler version 12.0.4.196. Our measurements record the wall clock time spent by each benchmark in *kernel* computation: i.e. without including the time taken to read input or write output. We take the average from a number of runs of each configuration of each benchmark.

All three GHC compilers (standard, LLVM, and our modified version) were invoked with the `-O2` option and the `-msse2` option. When using LLVM, we also passed the `-optlo-O2` option and the `-optlo-std-compile-opts` option to GHC. For certain benchmarks we have further tuned the optimization flags, usually according to suggestions provided by the benchmark authors. The same flags are passed to our modified GHC and to the standard GHC except in some limited cases where a flag was beneficial to GHC but not to HRC. To eliminate SIMD vectorization as a factor in performance, HRC was run without enabling the vectorization [21] pass. HRC/FLRC supports compilation with both a strict floating point model in which only value-safe IEEE compliant reductions are performed and in which source level precision is maintained; and a relaxed model in which non value-safe floating point optimizations (such as re-association) are performed, and in which the underlying C compiler is allowed to compute results using more or less precision than specified by IEEE semantics. For our benchmarks, all compilation was done with the strict floating point model.

All executables were run with a 1024 megabyte heap. For the GHC builds, this was done by passing the runtime arguments -*H1024m -M1024m*. This choice seemed to provide the best performance across a range of benchmarks, but was not highly tuned. The HRC executables were run with the same heap restrictions, and no

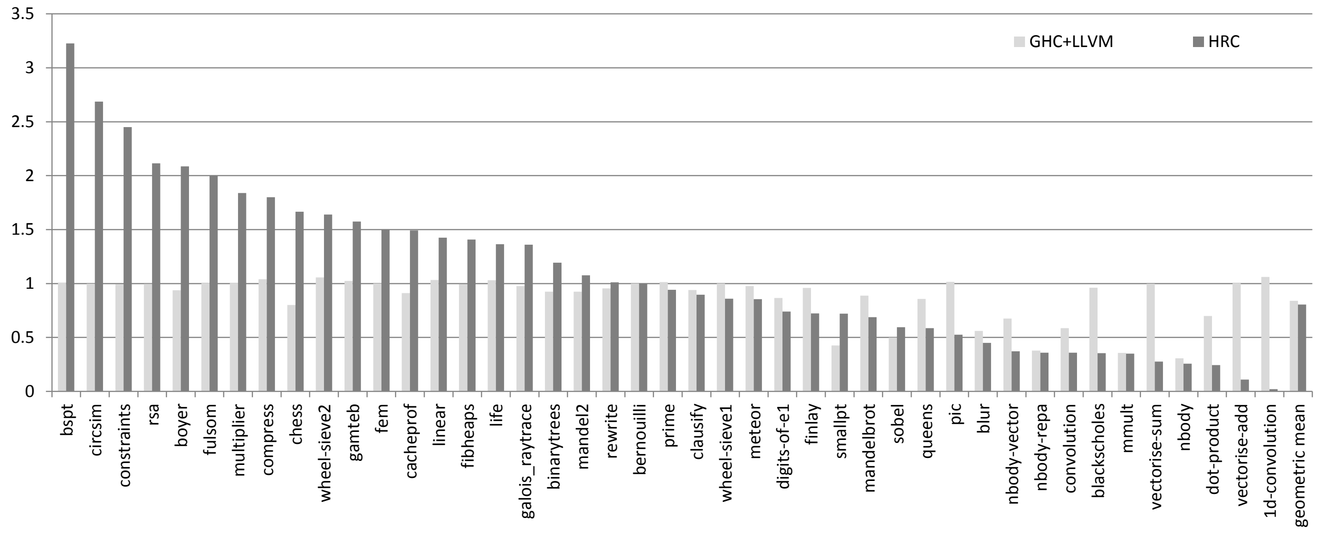

Figure 2. Kernel Execution Time Relative to GHC (smaller is better)

further tuning of the heap options was performed. For a few benchmarks, we required a larger stack size to be set at runtime than our standard default.

Figure 2 shows the comparison of normalized kernel execution time relative to standard GHC of all benchmarks. The normalized kernel time is computed by dividing the measured run time for a given configuration by the run time of standard GHC with its native backend (not LLVM). Lower is better on this graph, and performance parity with GHC corresponds to the value 1 on the y-axis. Bars are shown for each program as compiled by GHC with the LLVM backend, and by HRC. The benchmarks are sorted by the relative performance of the latter, which makes it clear which ones are worse than GHC, which are better, and by how much. Overall, the geometric mean of HRC is at parity when compared to the GHC LLVM configuration, which in turn is about 10% faster than the standard GHC with native backend.

We must also note that all GHC+LLVM performance numbers presented here were obtained from programs compiled by LLVM version 2.9 instead of a more recent version. This is because 2.9 is the only LLVM version that works reliably for all benchmark programs on 32-bit Windows. Using any other LLVM version from 3.0 to 3.3 would produce a segmentation fault error for a number of programs at runtime. For those that did run correctly, we noticed a 5% overall performance improvements in the geometric mean (relative to GHC with native backend) when LLVM 3.3 is used in place of LLVM 2.9.

To the left of Figure 2 are programs that perform better with GHC. Generally speaking, these tend to be programs written making extensive use of lists or other lazy data-structures that are difficult to make strict. Based on our qualitative analysis of the benchmarks, there seem to be a number of reasons why GHC outperforms our compiler on these benchmarks.

First and foremost the GHC runtime is highly tuned for executing lazy code and curried functions. Many of the programs on the left side of the graph are those for which HRC is unable to eliminate thunking and currying, result in higher-allocation (due to currying) or more overhead due to our more heavyweight thunk implementation. We have some quantitative evidence in particular for the latter in that benchmarks on the left side of the graph tend to be particularly sensitive to thunk-representation choice (we have several such choices). We believe that a more sophisticated approach to eliminating currying [5] than we have so far attempted might help. There is also some room for improvement in our thunk representation.

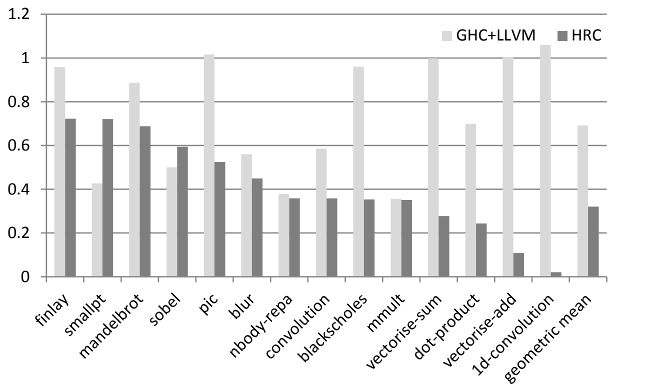

Figure 3. Kernel Execution Time Relative to GHC for Selected Benchmarks (smaller is better)

A second key performance differentiator between GHC and HRC on these benchmarks is the match between the allocation behavior and the underlying garbage collection approach. For a number of benchmarks on which we perform poorly, we observe that HRC compiled programs spend substantially more time in the garbage collector. This is partly due to using less efficient object representations in our runtime—our objects are larger and hence we allocate more and stress the GC more. However, even after accounting for this it is apparent that the design choices made by the GHC GC are much better suited to the allocation profile of many Haskell programs. The mark-sweep-compact algorithm used by the HRC TGC incurs substantial overhead when used with programs that allocate at such a high rate, with large amounts of fragmentation, and with relatively large live object counts.

On the right side of Figure 2 are programs on which HRC performs better than GHC. We present a selection of these separately in Figure 3. These benchmarks are generally performance-oriented Haskell programs. They include several example programs from the Repa examples package (such as blur and sobel image processing benchmarks), several computationally intensive mathematical kernels (e.g. matrix-mult, finlay), some small micro-benchmarks (e.g. dot-product, vectorise-add and vectorise-sum), some general throughput oriented benchmarks (e.g. nbody, convolution, 1d-convolution), and a variety of other computationally intensive benchmarks. The geometric mean for this selected group shows that HRC is about 2× faster than GHC with LLVM, and 3× faster than standard GHC.

4.1 Performance analysis

It is difficult to quantify contributions of specific optimizations to benchmark performance, since almost all optimizations interact synergistically with others. Nonetheless, we believe that interesting insights into the contribution of the various optimizations can be obtained by selectively eliminating one optimization (or set of optimizations) at a time, and measuring the resulting performance penalty. We have performed a series of such experiments using the subset of the benchmarks chosen for Figure 3 plus the galois_raytracer benchmark. In the following discussion, we have measured performance with an optimization removed and added back in, and report the percent speedup of adding back in.

The backend C compiler optimizations are crucial for performance. Comparing no optimization to full optimization, we observe speedups ranging from 41% and 91%, with a geometric mean across the benchmarks of 70%. Much (but not all) of this benefit can be obtained with the simple -O1 level optimizations. Comparing this level of optimization to full optimization we observe speedups ranging from -6.5% to 32%, with a geometric mean of 8%. Clearly there are substantial benefits provided from the full level of optimization, but on average it seems that more limited optimization and code generation can provide adequate performance.

Comparing the compiler with the entire suite of MIL optimizations disabled to the standard configuration, we observe that the MIL optimizations provide between a 20% and 98% speedup, with a geometric mean of 75%. We break this down further by disabling various of the specific optimizations within the MIL pipeline discussed in Section 2.2. Because the MIL optimizations are highly synergistic, these experiments are somewhat harder to interpret, but nonetheless interesting. The contification optimization provides a speedup ranging from 0% to 97%, with a geometric mean of 62%. That these numbers are close to the speedups obtained by the entire MIL pipeline reflects in part the fact that the contification optimization is a crucial enabling optimization for all of the MIL optimizations. The flow analysis based representation optimizations provide between a -6.1% and 97% speedup, with a geometric mean of 25%. The experimental thunk optimizations discussed at the end of Section 2.2 provide a speedup of between -12% and 97%, with a geometric mean of 20%. The loop invariant code motion pass provides only small benefits, ranging between -1.9% and 5.9%, with a geometric mean of 0.32%.

Our experimental strictness analysis on the ANormLazy representation provides us speedup between -3.9% and 11%, with a geometric mean of 0.82%. The ad hoc strictness at the ANormStrict level provides between -6.7% and 97% speedup, with a geometric mean of 23%. We suspect that the significantly better speedups provided by the ad hoc strictness relative to our ANormLazy strictness most likely reflect its position in the phase ordering after other simplifications have been performed, but do not have strong evidence for this. The uncurrying optimization in the ANormStrict optimizer provides between a -5.3% and a 29% speedup, with a geometric mean of 5.6%. Unfortunately, an outstanding compiler bug prevents us from fully disabling the remaining ANormStrict optimizations for measurement.

These numbers provide some indication of the relative importance of the various components of the compiler pipeline. Some optimizations clearly play a crucial role in achieving any performance at all with our stack, while others contribute significantly to certain benchmarks and not at all to others. Qualitatively, we have observed that our compiler is often able to make small but crucial improvements to key inner loops in these programs that result in significant performance gains. Examples include the elimination of uses of laziness, improved representations for runtime data-structures (e.g. unboxing), hoisting of code out of loops, and elimination of unnecessary branches. For most of the programs in Figure 3, our compiler is able to turn the performance critical sections into almost entirely local control-flow, for which our compiler is well-tuned. For certain of these benchmarks (notably the 1d-convolution benchmark), it is striking the extent to which disabling any one of a number of optimizations eliminates almost all performance improvements from other optimizations.

The performance of many programs included in Figure 3 can be significantly further improved by HRC using auto vectorization [21] on SIMD-capable hardware. We have been careful not to include this optimization in our performance study here, since we wish to focus on establishing a baseline sequential comparison.

While the overall performance results achieved so far are mixed, we believe that the HRC experiment provides valuable data about tradeoffs and opportunities that lie in the different design choices available to compiler implementers. We also believe that this experiment suggests that the limits of Haskell performance have not been reached by existing compiler technology.

4.2 Compile Time

Our compiler has not been engineered for compilation time, and there are numerous known opportunities to speed up its performance. However, design choices were made with the intention of providing good scalability up to very large programs. While many of these benchmarks are textually small, they pull in very large sets of libraries that must be compiled by HRC in whole. If we consider a pretty-printed Core IR (after dependence analysis and pruning of unused code) as input, on this set of benchmarks, the program size varies between 50k to 180k LOC (lines-of-code), and compile time ranges from 1 minute and 34 seconds to 9 minutes. Summed over all of these benchmarks, approximately 27% of the compile time was spent in the frontend passes (including the GHC frontend and those labeled HRC in Figure 1), 49% was spent in MIL passes, 22.5% in the backend Pillar and C compilers, and 1.5% in the linker.

5. Discussion

We did not set out to write a Haskell compiler, but came from the perspective of adapting an existing functional language compiler, hoping for an interesting experiment to see if our separately developed techniques could be applied to Haskell. Many of the choices that we made, were made in the context of that previous language, and were not made because we thought they would be best for Haskell. But by this experiment, we get to see to what extent they are, or are not, reasonable choices for compiling Haskell.

Reusing GHC was clearly a big win. The effort involved in building a lexer, parser, and type checker for Haskell is immense, not to mention some sort of reasonable standard library. GHC has all this plus high-level optimizations and can output a small intermediate representation. Our experience with reusing GHC was mostly positive. External Core is indeed easy to use as the starting point for a Haskell backend. GHC's primitives are not so straightforward to implement and some impedance matching is necessary. Apart from the known limitations described in section 3.4, HRC is able to compile and correctly run most, if not all, nofib benchmarks, as well as a good portion of GHC testsuites. Popular Haskell libraries such as repa, parsec, monad-par, criterion, etc., are also supported with little or no modification.

We chose to build a whole-program compiler based mostly on an SSA-based CFG-based intermediate representation and optimizations. This choice was inspired by MLton [30], which showed the benefits of that approach for functional languages, and we believe that we benefit from some of these advantages. We chose to use a high-level object model based on initializing writes in this low-level of control-flow representation. We have many optimizations that exploit immutability properties and that in combination with traditional loop optimizations can do things that optimizations at higher levels of representation cannot. Our optimizations are not meant to replace those at a higher level of representation, and we clearly benefit from the high-level optimizations of GHC; instead they are complimentary, and our performance data clearly show they can be very beneficial in some classes of applications.

We chose, for our previous language, to use a conventional runtime and object model, and not to tailor their design to the language. We decided to stick with this choice for Haskell, in contrast to GHC's STG machine and GC. Our experience with these choices was mixed. In many Haskell programs we can overcome the overheads of not using a tailored runtime and GC, but for some, we clearly suffer compared to GHC. Our GC was developed for a strict, mostly pure, functional language, and works well in that context. Haskell, however, from the perspective of garbage collection is not mostly pure, mutating heavily if laziness is used extensively. We had some previous experience with a GC for Java, but Haskell, in contrast to Java, also has a high allocation rate. Thus our experience indicates that the high mutation and high allocation rates mean that choices that might work well for other languages do not work well for lazy functional languages. GHC clearly has made a careful set of design choices for its GC and dramatically out-performs our GC on a number of programs. While we do not have direct evidence, we also suspect that when extensive currying and partial application are used, GHC's STG machine approach has substantial benefit.

We chose to use Pillar to separate our compiler from low-level code generation. Pillar, like C−−, is intended to provide high-level language implementers with a target that is portable and handles issues like register allocation, instruction selection, instruction scheduling, and optimization for the target architecture. On modern platforms such issues are important to address well, and take considerable effort to do and to do well, and that effort has to be repeated anew for each target platform. An infrastructure like C−−, Pillar, or LLVM is a big win for high-level language developers.

The original vision for Pillar was to support several high-level languages and we originally implemented a native compiler for Pillar. That vision never materialized, and for completely non-technical reasons we abandoned our native Pillar compiler and wrote a converter to C. We learned two lessons from that experience. First, while converting to C has overheads and does not perform as well as a native compiler, those overheads are not that high. Second, we benefit a lot from using the Intel C compiler. Lots of effort goes into the code generation part of the Intel compilers, and the knowledge of our processors and their microarchitectures informs the low-level optimizations. Furthermore, the compiler continually tracks newer versions of our processors, and provides us with the performance benefits available from specifically targetting the processor being used. We observe that GHC gets similar benefits from using the LLVM infrastructure.

In summary the lessons we learned are:

- Reusing GHC as a frontend is a good idea. External core is easy to use. Reusing GHC's libraries is doable, but less easy.
- Low-level control with high-level object model representations exploiting knowledge and invariants of the high-level language provides benefits that functional languages implementers should consider.
- Separating allocation from initialization using initializing writes is a powerful technique for lowering immutable objects to a lower level where additional optimizations and transformation can be applied.
- The overheads of not using a specialized runtime such as the STG machine and GHC's GC can be overcome on many Haskell programs, but are important to some.
- Eliminating thunks from hot loops is critical to achieving high-performance for Haskell programs.
- An infrastructure for separating high-level language implementation from low-level code generation is very beneficial for high-level language implementers.
- There are overheads to compiling through C, but with careful design these can largely be overcome. In turn, the benefits provided by the industrial strength code generation of modern C compilers such as Intel's can be very substantial. An Intel compiler for Pillar or C−− would obviously be preferable.

6. Related and Future Work

Besides GHC, there are a number of other compilers and/or interpreters for Haskell including UHC [7], JHC [17], and a few others that are no longer maintained.

UHC supports most of Haskell 98 standards with some extensions. It also employs multiple backends, including an interpreter, a whole-program compilation backend called GRIN (Graph Reduction Intermediate Notation) that eventually outputs machine code, and some other ones including a Javascript backend. UHC uses a heap "point-to" analysis on GRIN to eliminate unknown control flow due to thunk evals. The MIL IR used by our compiler is at a slightly lower level than GRIN because it is based around explicit basic blocks. UHC is also known for its novel use of Attribute Grammar (AG) and an aspect oriented internal organization, while we take a more traditional multi-pass and multi-IR compiler approach. JHC is another Haskell compiler with many experimental features including a unique class implementation and region inference among others. It also uses a variant of GRIN as one of its intermediate representations.

Both UHC and JHC aim to compile Haskell from source with their own implementations of type analysis, Haskell extensions, high level transformations, etc., and thus they are not fully interoperable with GHC, Haskell's defacto standard implementation.

GHC itself has gone through a lot of changes over the years, gaining a highly-tuned runtime and sophisticated garbage collector, and a LLVM backend, among others. The *Strict Core* proposal for GHC [6] unfortunately was not implemented in GHC's main branch due to its potential impact to the already complicated system. We make use of both a lazy and a strict ANorm IR, and the latter bears many similarities to the Strict Core.

GHC's native backend translates from Core to STG, and then to Cmm, a variant of C−−, which was designed to be "portable assembly" that eases translation from high-level languages to machine code. It has a simple machine-level type system, supports tail calls, and has interfaces for garbage collection and exception handling.

While C−− strives to be small, simple, and portable, LLVM aims to be comprehensive, multi-purpose, and portable. Due to its large collection of tools and ease of use, LLVM is becoming a popular choice among compiler writers. Even GHC has a LLVM backend that translates from Cmm to LLVM's IR. LLVM's IR is control flow and SSA based, which is indeed very similar to MIL except that LLVM IR is more assembly like, and MIL is at a slightly higher level. While the LLVM IR also maintains static type information, MIL has more elaborate meta-data and types, as well as effect annotations. LLVM now also supports GC implementation through a compiler plug-in. Terei and Chakravarty give a more detailed comparison between Cmm and LLVM [28].

There are also other high-level virtual machines such as Microsoft's Common Language Runtime (CLR) and the Java Virtual Machine (JVM) that provide portable and high-performance compiler backend. Some other functional languages, e.g., Scala and Closure, have been targeted to these virtual machines rather than to real hardware. These virtual machines usually provide certain modern features such as memory safety and GC, but they have abstracted away many hardware features to achieve portability. HRC was designed to make best use of Intel hardware through both high-level and low-level optimizations, and hence we have not considered targeting CLR or JVM.

Aside from focusing on sequential performance in compiling lazy languages, we have also experimented with SIMD parallelism through auto vectorization [21] and looked at performance for multicores and Intel's Xeon Phi co-processor [20]. Our future work will continue to investigate these topics and exploit more hardware features such as integrated GPUs.

7. Conclusion

Being one of the most advanced functional-language compilers, GHC is hard to beat in terms of its feature set, performance, and robustness. By leveraging GHC itself as a frontend, we take advantage of GHC's high-level optimization before Core, a lazy IR, and then we focus our effort on compiling Core to MIL, a strict IR. Through a multitude of aggressive optimization passes, we produce good-quality low-level imperative code for performance-oriented programs, and overcome the lack of a specialized runtime for a lazy language. Along the way, we have learned many lessons about the pros and cons of various design and implementation choices, and demonstrated that a good compiler can achieve native machine-level performance for functional programs typically composed through high-level abstractions. Properties of functional languages such as type safety and immutability by default are also crucial to many of these optimization techniques, not easily obtainable in compiling traditional imperative languages. We hope our descriptions are useful to future Haskell implementers, and provide them with options to consider. We also hope that our data demonstrates that the last word on Haskell performance is yet to be said.

References

[1] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *ISMM*, pages 21–30. ACM, 2010.

[2] T. A. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *LCPC*, volume 5234 of *LNCS*, pages 141–155. Springer-Verlag, 2007.

[3] A. Appel and T. Jim. Shrinking lambda expressions in linear time. *JFP*, 7(5), Sept. 1997.

[4] N. Benton, A. Kennedy, S. Lindley, and C. Russo. Shrinking reductions in SML.NET. In *IFL 2004*, volume 3474 of *LNCS*, pages 142–159. Springer-Verlag, 2005.

[5] L. Bergstrom and J. Reppy. Arity raising in Manticore. In *IFL 2009*, volume 6041 of *LNCS*, pages 90–106. Springer-Verlag, 2010.

[6] M. C. Bolingbroke and S. L. Peyton Jones. Types are calling conventions. In *Haskell Symposium*, pages 1–12. ACM, Sept. 2009.

[7] A. Dijkstra, J. Fokker, and S. D. Swierstra. The architecture of the Utrecht Haskell compiler. In *Haskell Symposium*, pages 93–104. ACM, Sept. 2009.

[8] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *PLDI*, pages 237–247. ACM, June 1993.

[9] M. Fluet and S. Weeks. Contification using dominators. In *ICFP*, pages 2–13. ACM, Sept. 2001.

[10] N. Glew and L. Petersen. Type-preserving flow analysis and interprocedural unboxing (extended version), Mar. 2012. arXiv:1203.1986 [cs.PL], `http://arXiv.org/`.

[11] N. Glew, T. Sweeney, and L. Petersen. A multivalued language with a dependent type system. In *Dependently Typed Programming*. ACM, Sept. 2013.

[12] K. Jensen, P. Hjresen, and M. Rosendahl. Efficient strictness analysis of Haskell. In *Static Analysis*, volume 864 of *LNCS*, pages 346–362. Springer-Verlag, 1994.

[13] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. *ACM Sigplan Notices*, 45(9):261–272, 2010.

[14] S. Marlow and S. Peyton Jones. The new GHC/Hugs runtime system. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=68449`. Jan. 1998.

[15] S. Marlow and S. Peyton Jones. Making a fast curry: Push/enter vs. eval/apply for higher-order languages. *JFP*, 16(4-5):415–449, July 2006.

[16] S. Marlow and S. L. Peyton Jones. Multicore garbage collection with local heaps. In *ISMM*, pages 21–32. ACM, June 2011.

[17] J. Meacham. JHC: John's Haskell compiler, 2007. URL `http://repetae.net/computer/jhc/`.

[18] W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, Workshops in Computing, pages 195–202. Springer-Verlag, 1993.

[19] L. Petersen and N. Glew. GC-safe interprocedural unboxing. In *CC*, volume 7210 of *LNCS*, pages 165–184. Springer-Verlag, Apr. 2012.

[20] L. Petersen, T. A. Anderson, H. Liu, and N. Glew. Measuring the Haskell gap. Manuscript available from the authors, June 2013.

[21] L. Petersen, D. Orchard, and N. Glew. Automatic SIMD vectorization for Haskell. In *ICFP*. ACM, Sept. 2013.

[22] S. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In *Functional Programming, Glasgow 1993*, Workshops in Computing, pages 201–221. Springer-Verlag, 1994.

[23] S. Peyton Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In *PPDP*, volume 1702 of *LNCS*, pages 1–28. Springer-Verlag, Oct. 1999.

[24] S. Peyton Jones, A. Reid, F. Henderson, T. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *PLDI*, pages 25–36. ACM, May 1999.

[25] S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *JFP*, 2(2):127–202, Apr. 1992.

[26] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, pages 717–740. ACM, 1972.

[27] M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, pages 53–66. ACM, Jan. 2007.

[28] D. A. Terei and M. M. Chakravarty. An LLVM backend for GHC. *ACM Sigplan Notices*, 45(11):109–120, 2010.

[29] A. Tolmach. An external representation for the GHC Core language. URL `http://www.haskell.org/ghc/docs/papers/core.ps.gz`. Sept. 2001.

[30] S. Weeks. Whole-program compilation in MLton. In *ML Workshop*, pages 1–1. ACM, Sept. 2006.

[31] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *MICRO*, pages 1–11. IEEE, Nov. 1994.