Low-Level Compiler Optimizations for Functional Programming Languages

David M. Peixotto, Keith D. Cooper
Rice University / Computer Science

Problem

Abstract
Functional programming languages provide the programmer with a high-level declarative interface for describing algorithms. This high-level interface poses challenges to efficiently compiling these languages for execution on traditional machines. Many years of research have been invested to bring the efficiency of these high-level languages closer to that of traditional complied languages, such as Fortran and C. Much of this past work relies on high-level transformations that are enabled by the functional semantics of the input language. In this research, we investigate how traditional compilation techniques can be leveraged to improve the performance of high-level functional programming languages.

We propose a research agenda to throughly investigate the effectiveness of low-level compiler optimizations for high-level lazy functional languages. In this research, we seek to answer the following questions. To what extent can we leverage traditional low-level compiler optimizations in a high-level functional programming language? Can we identify the high-level transformations that expose more opportunities for low-level optimizations and selectively apply the transformations to enable more optimization opportunities in the back end of the compiler? Are there opportunities for new low-level optimizations that are enabled by operating in the context of translation from a high-level functional programming language?

High-Level Optimizations

Inlining (Context Transformation)
Replace an occurrence of a definition with its body. Unconditionally valid in a lazy purely-functional language.

Dead Code Elimination (Computation Transformation)
Delete an unused binding. Unconditionally valid in a lazy purely-functional language.

Let Floating (Code Motion Transformation)
Move let bindings (thunk allocations) down code branches to reduce the number of allocations, or move them out of lambda expressions to increase sharing (full laziness).

Case Transformations (Control Flow Transformation)
Case of known constructor can eliminate unused branches of a case statement. Case of case transformation moves an outer case inside an inner case. Inlining and dead code can then eliminate useless bindings.

Strictness (Memory Usage Transformation)
See which arguments to a function will be used and evaluate them eagerly rather than allocating a thunk for later evaluation.

... and many more

Experimental Platform

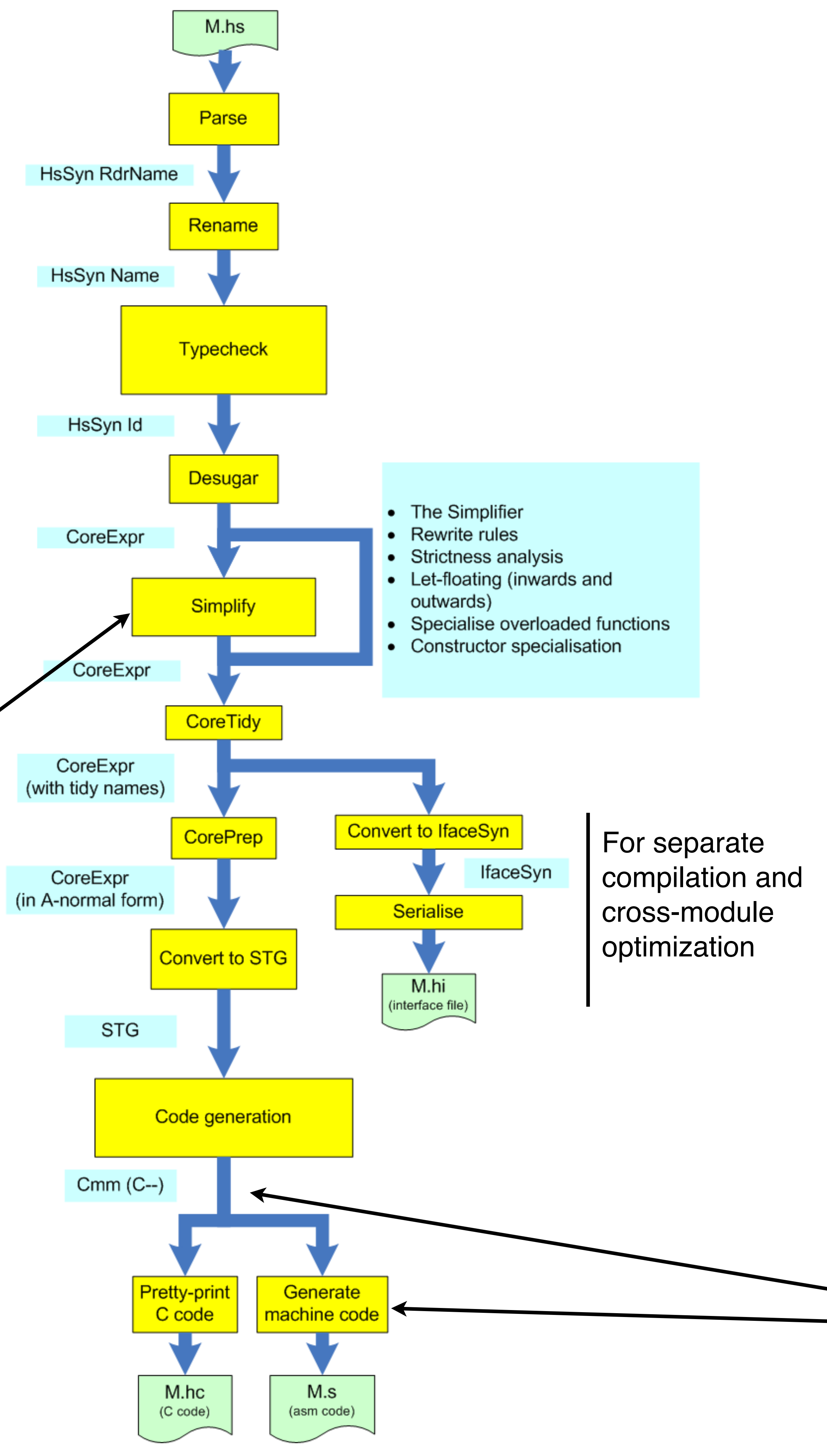
Key Investigations
1. To what extent can we leverage traditional low-level compiler optimizations in a high-level functional programming language?

2. Can we identify high-level optimizations that expose more opportunities for low-level optimizations.

3. Are there opportunities for new low-level optimizations that are enabled by operating in the context of translation from a high-level functional programming language.

4.Can we utilize adaptive compilation to selectively optimize the program at a high-level in order to enable more optimization opportunities in the back end of the compiler.

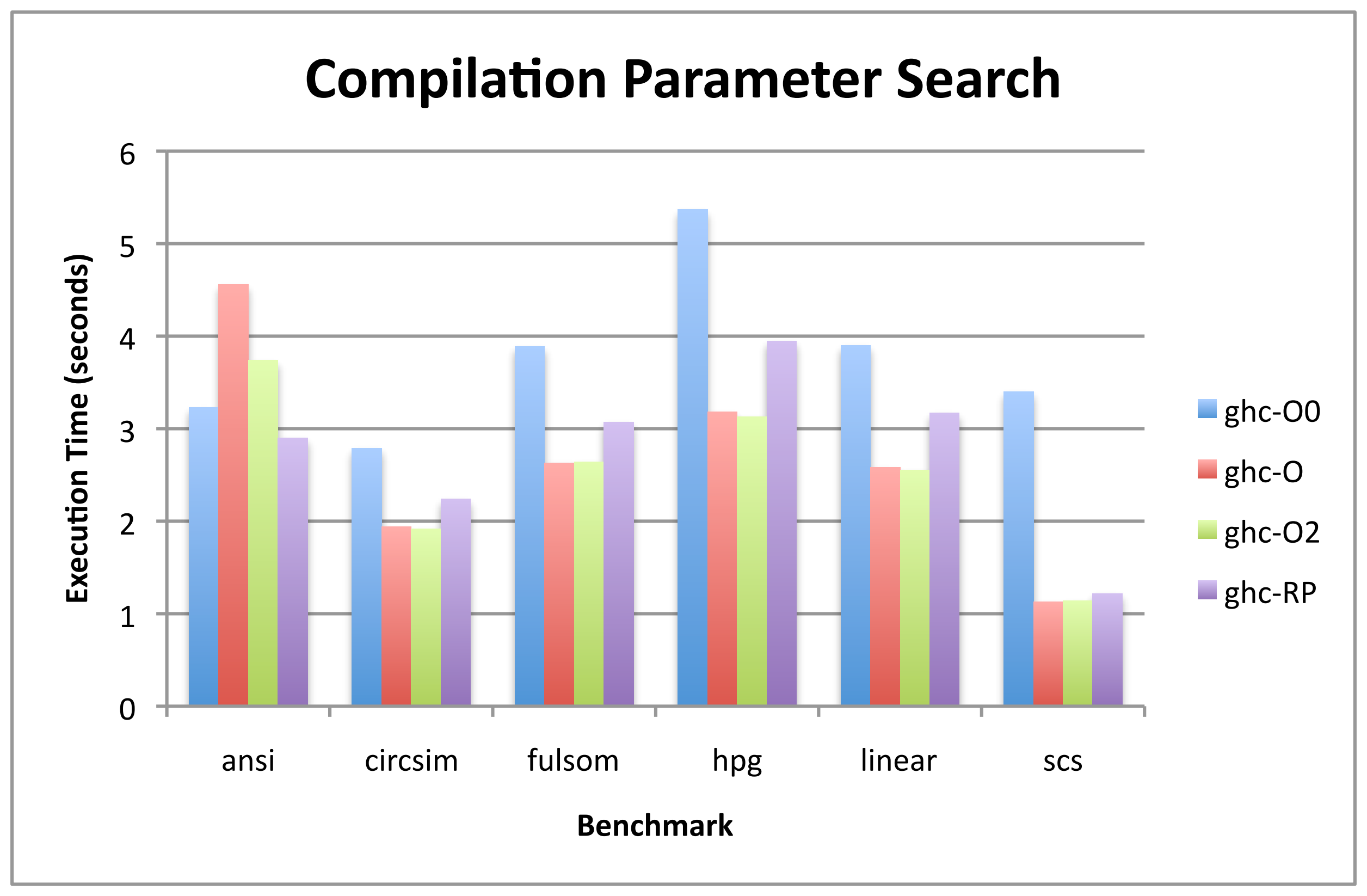
Compilation Pipeline



Approach

Thesis
Traditional compiler optimizations can be used to improve the performance of a compiled high-level functional programming language. The opportunities for applying these low-level optimizations will depend heavily on the shape of the input code, which is influenced by the high level transformations done in the earlier stages of the compiler. Using adaptive compilation techniques to selectively apply high level optimizations, we can effectively enable opportunities for low-level optimizations and see an overall increase in performance.

Adaptive Compilation



Idea
Search for good combinations of compiler parameters to expose optimization opportunities to the back-end compiler.

Methodology
Select compiler flags from a set of 26 boolean and integer parameters. Use *random probing* to select and evaluate combinations of parameters.

Results
Adaptive compilation shows promise as a technique for exposing optimization opportunities. Better search algorithms and back-end optimizations are needed to fully understand the potential.

Low-Level Optimizations

Current
1. Mini-inliner – simple forward propagation for once-used values
2. Simple constant folding – simply instructions when one argument is a constant
3. Loopify – change tail-recursive function to a loop
4. Register allocation (Linear scan and Graph coloring)

Under Consideration
1. Traditional global optimizations (e.g. value numbering)
2. Loop optimizations for DPH (e.g. unrolling, SW pipelining)
3. Lightweight concurrency (cf. Peyton Jones and Ramsey)
4. Register allocation (e.g. ELS with coalescing and SSE optimizations)