Reducing Compilation Latency in the Julia Programming Language

by

Prem Chintalapudi

S.B. Biological Engineering and Computer Science and Engineering,
Massachusetts Institute of Technology, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Prem Chintalapudi
Department of Electrical Engineering and Computer Science
June 2023

Certified by: Alan Edelman
Department of Mathematics
Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Reducing Compilation Latency in the Julia Programming Language

by

Prem Chintalapudi

Submitted to the Department of Electrical Engineering and Computer Science
on June 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Julia programming language is a high performance computing language that employs an LLVM-based just-in-time compiler and an LLVM-based ahead-of-time compiler to produce optimized machine code. When Julia uses its just-in-time compiler, compilation of methods must be done before methods can begin execution, which presents as a delay the first time a function is executed. When Julia compiles code ahead of time into code images, a large fraction of time is spent optimizing and emitting machine code for large numbers of functions. Here, we investigate ways of exposing opportunities for parallelism to both compilers, as well as investigate ways to perform less work during the compilation process using newer LLVM technologies. Our results show that we can achieve speedups of 8-16X in the ahead-of-time compiler when compiling on multiple threads, while the just-in-time compiler can achieve speedups between 1.5-3X under certain circumstances. Additionally, we find that LLVM's new CompileOnDemandLayer for delaying compilation until code is executed can avoid 30-40% of compilation work in certain applications, while LLVM's new pass manager framework can reduce optimization time by up to 17.5% compared to the legacy pass manager. Incorporation of these compiler improvements into the Julia language yields marked decreases in the initial delays that are observed by users today.

Thesis supervisor: Alan Edelman

Title: Department of Mathematics

Acknowledgments

I would like to express a very big thank you to the entirety of the JuliaLab and JuliaHub, as well as the Julia community at large, for all of their support and advice through this process.

In particular, I would like to thank Valentin Churavy, for his extremely pertinent and helpful insights, as well as his overall guidance and mentorship. This work would not have been possible without him, and I greatly appreciate all that he has done.

I would also like to thank Jameson Nash, who has spent much time reviewing my work. Jameson and the rest of the JuliaHub team have been crucial to making my contributions valuable, and I thank them for that.

I also appreciate Prof. Alan Edelman, who has provided assistance freely throughout my time with the JuliaLab.

Finally, I thank my family and friends for their invaluable kindness and encouragement. Their support makes the work I do worthwhile, and is irreplaceable to me.

Contents

List of Figures

Chapter 1

Introduction

The Julia programming language is a high-performance dynamically-typed JIT-compiled language well suited for scientific computation workloads. The language's stated goal is to solve what it characterizes as the "two-language problem," where one programming language (such as Python) is used to write simple glue code for libraries written in another, higher performance but lower-level language (e.g. C++). To achieve its performance requirements, Julia embeds a type-inference system as well as an optimizing compiler. Julia's optimizing compiler is itself split into two pipelines, one of which is written in Julia and another of which is written using LLVM, a collection of compiler utilities.

One of the more unique features about Julia is its use of multiple dispatch and heavy use of generic programming. When a method is called in Julia, the runtime will inspect the concrete types of the arguments to that method and select the method definition with the best match to those arguments. If that method definition has never been compiled for the passed in types, the compiler will perform type inference and optimization before emitting machine code to memory. Once this machine code emission has completed, execution will be continued, calling the fast compiled method. Subsequent calls to the same method with the same types will not result in compilation, as the calls will dispatch to the same compiled machine code.

Currently, Julia faces a problem known variously as time-to-first-plot (TTFP), time-to-first-execution (TTFX), or compilation latency. Briefly, this is the time that is

spent inferring and compiling Julia source code to machine code that can be executed on a CPU directly. As described above, before a method may be executed for the first time, it must be compiled. However, the types that a method will be called with are known only after that method has been called. Thus, the execution time of the first call to a method includes compilation time, and a slow compiler results in slow performance measurements for methods that are only called once.

Here we show that we can improve the performance of the compiler drastically by introducing parallelism in the time-consuming portions of the compiler. We also find that new LLVM features such as lazier compilation tools and a new optimization pass manager significantly reduce the amount of compilation work that must be done, further reducing TTFX measurements.

Chapter 2

Background

2.1 LLVM

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies [13]. Julia primarily relies on LLVM's intermediate representation (IR), a method of representing source code in a generic and predictable manner. LLVM IR is a static-single assignment (SSA) IR, which means every variable is assigned once and only once in a program. Furthermore, the IR is broken up into several levels of structure, starting with modules which contain one or more functions, functions which contain one or more basic blocks, basic blocks which contain one or more instructions, and instructions which map very closely to corresponding assembly instructions. SSA-form IR enables a wide variety of optimizations to be represented as simple replacements of one value with another.

LLVM also provides an optimization pass manager as well as several optimization passes which can be used to canonicalize and simplify the IR for a faster execution time. However, while the final code that is produced is fast and efficient, producing LLVM IR and optimizing it are slow steps and contribute significantly to compilation latency.

2.2 JIT Compilation

JIT compilation, or just-in-time compilation, is the process of generating code and subsequently executing this generated code during the process of program execution. JIT compilation is often contrasted with ahead-of-time (AOT) compilation, where a compiler produces an executable that is then run as the actual program. Compared to AOT compilation, JIT compilation includes compilation overhead during the running of the program, which takes time and resources away from the actual execution of the user's code. However, AOT compilation can only optimize code based on knowledge that is available prior to execution of a program, while a JIT compiler is free to run compilation at a time of its choosing during the execution of the program, when more information is available.

In Julia's case, methods are specialized for every combination of types [6]. In an AOT compiler, this would necessitate specializing a method for every possible combination of argument types, which is infeasible to compile due to combinatorial explosion. However, Julia gets around this problem by only compiling for particular combinations of arguments that are passed in at runtime, which necessitates a JIT compiler to generate machine code.

2.3 Julia's Compilation Process

Julia embeds a number of different stages in its compilation process, as shown in figure 2-1.

2.3.1 In-Julia Compilation

First, any macros are expanded and code is lowered into an abstract syntax tree (AST), a representation that removes order of operations and syntactical rules from the code. Then, the AST is converted to a Julia SSA form (Julia IR). This Julia IR is then run through an optimization pipeline to initially simplify the code, before it is piped to the LLVM IR generator (known as codegen).

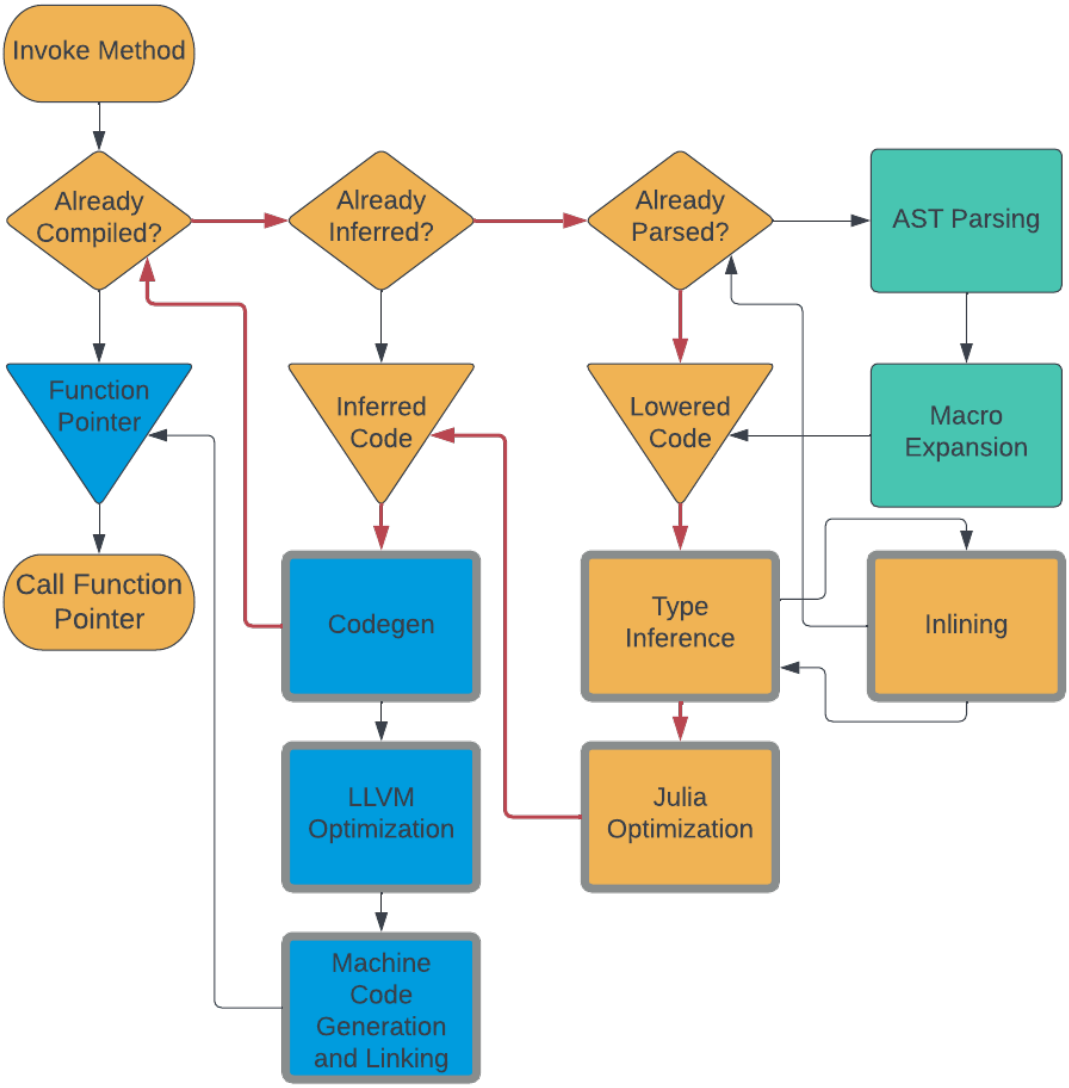

Figure 2-1: Julia Compilation Pipeline

Julia's compilation pipeline passes source code through a variety of stages to parse, transform, and optimize the program for a faster execution time. Steps shaded in blue are done using LLVM and written in C++, steps shaded in yellow operate only on Julia data structures and are written in Julia, and steps shaded in green are written in femtoLISP, a custom dialect of LISP. Steps with a thick gray outline are performed only while the global compilation lock is held. Thick red lines indicate steps that may trigger additional functions to be compiled or inferred.

2.3.2 LLVM Compilation

All of the above steps are written in Julia itself. For codegen and subsequent steps, however, the implementation is written in C++ as LLVM APIs are defined in terms of C++.

Codegen

Codegen is the process of taking Julia IR and converting it to LLVM IR. LLVM IR, while superficially similar to Julia IR in that both are SSA-form IRs, offers a much more expressive transformation API at the cost of slower compilation speed. LLVM IR also has a more mature optimization ecosystem, and contains facilities to emit machine code, both of which are lacking for Julia IR.

LLVM IR relies on the presence of an LLVM context, which is an object that contains state common to various IR constructs. Operations on different context objects are thread safe, but operations on the same context are not. Parallelization of LLVM code generation and optimization must therefore move or duplicate LLVM IR into different contexts in order to effectively operate on LLVM IR from multiple threads.

As the final step of codegen, once all of the relevant methods have been located and compiled to their own individual modules, the method modules are merged into a single module that contains all of the methods being compiled in this batch and emitted to the ORC JIT.

ORC JIT

On-Request Compilation (ORC) JIT is LLVM's JIT compilation toolchain, which supports in-memory compilation and linking. ORC is designed to support building custom JIT compilation stacks by composing ORC layers. An ORC layer is a collection of resources that are used to transform some input information into a final compiled output. A typical implementation of an ORC layer transforms its input of type A into an output of type B, then emits the output of type B to some base ORC

layer that takes an input of type B. This allows layers to form a JIT compilation stack, where layers can be added and removed freely to customize the JIT.

Julia's JIT compilation stack is depicted in figure 2-2, and includes an optimization layer, a compilation layer to generate object files, and a linker layer to make those machine code object files available for execution.

2.4 Parallel Speedup Measurement

In addition to knowing about the compilation pipeline, it is also useful to understand how performance of a program is assessed. While the definition of how much a serial optimization helped is quite trivial (% decrease in runtime/memory usage), parallelism optimizations are more complicated to analyze.

Measuring parallel speedups is typically performed in one of two different ways: by measuring how much time a fixed workload takes as additional threads are added, or by adding both additional threads and additional work and measuring the additional time taken. These two measurements are referred to as strong scaling and weak scaling, and are described by Amdahl's law and Gustafson's law, respectively.

2.4.1 Amdahl's Law

Amdahl's law [4] describes how the speedup of a task is a function of the parallel fraction of the total work and the number of processors allocated to the task. Furthermore, Amdahl's law assumes that as processors are added, the work remains equally divisible among the new number of processors. Here, we present a short derivation of the law based on these principles, where T_1 is the time taken for single-thread serial execution, T_P is the time taken for execution on P processors, and p is the fraction of work that is parallelizable.

$$T_1 = T_1((1-p)+p) \tag{2.1}$$

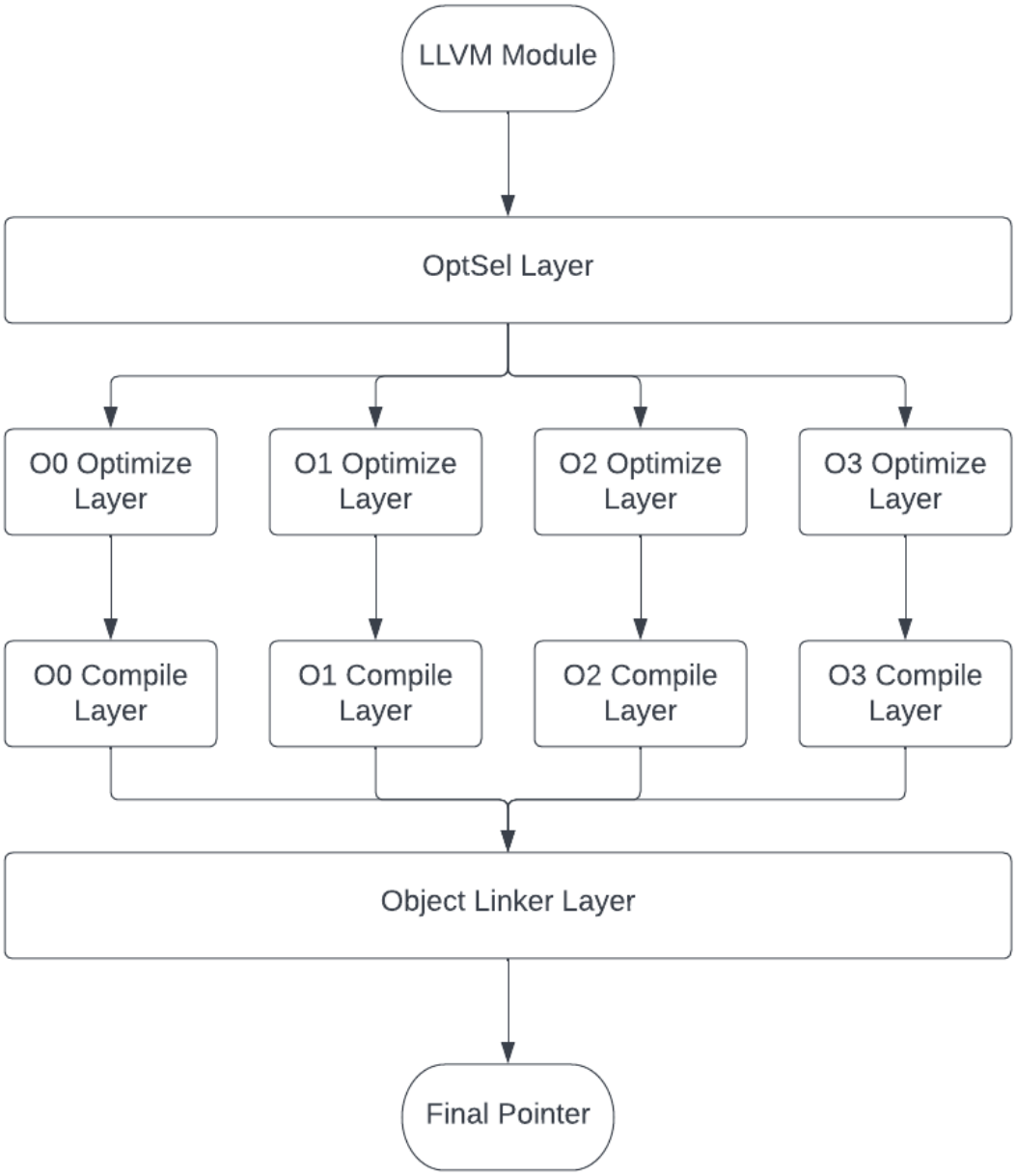

Figure 2-2: Julia JIT Compilation Stack

Julia's JIT stack first determines an optimization level for the current module, then dispatches modules to the appropriate optimizer and compiler, before linking the module's functions and globals into the current process. The function pointer may be obtained once linking is complete.

$$T_P = T_1(1-p) + T_1\frac{p}{P} \tag{2.2}$$

$$S_P = \frac{T_1}{T_n} = \frac{T_1}{T_1(1-p) + T_1\frac{p}{P}} = \frac{1}{(1-p) + \frac{p}{P}} \tag{2.3}$$

Amdahl's law suggests that the speedup gains from additional processors decrease with every additional processor added. In the limiting case of infinite processors, the maximum speedup predicted by Amdahl's law comes out to $\frac{1}{1-p}$, which is the reciprocal of the fraction of the workload that is not parallelizable. Therefore the end result of Amdahl's law is to suggest that parallel efficiency gains come with a cap for a fixed workload.

2.4.2 Gustafson's Law

From a user's perspective, however, increases in efficiency and speedups are typically followed by running larger workloads than before. Gustafson's law [10] addresses this by noting that the larger workloads typically increase the work in the parallel section without increasing the work in the serial section. In such a case, increasing processor count permits an arbitrarily larger amount of work to be accomplished in the same amount of time. Gustafson's law is defined as $S_P = (1-p) + pP$, to show that the effective speedup increases without bound as the number of processors increases. It does however assume that the serial workload does not increase even as the parallel workload increases, which may not always be a valid assumption for the problem.

Chapter 3

Related Work

3.1 Overview

The problem of reducing compilation latency has been addressed previously in a number of ways over the last 30 years. However, method invalidation poses a new problem to Julia compilation in particular, and this section is devoted to analyzing prior work in the field as well as the likely impact of method invalidation on those strategies.

3.2 Method Invalidation

One of Julia's core features is the ability to redefine compiled methods at runtime, a feature supported by few other compiled languages. As an example, in figure 3-1, method f is originally defined to return 5, but is then redefined to return the empty string. However, since method g used method f, method g must also be recompiled to call the new version of function f. In other words, both methods f and g were invalidated by the redefinition of f [12].

The implementation of invalidation is handled by storing a list of backedges to dependent methods inside the relevant method's methodinstance definition; in figure 3-1, g would be registered as a backedge of f. During compilation, every method that was called by g would have g appended to its backedge list, forming a reverse call graph

```
julia> f() = 5
f (generic function with 1 method)

julia> g() = f()
g (generic function with 1 method)

julia> g()
5

julia> f() = ""
f (generic function with 1 method)

julia> g()
""
```

Figure 3-1: Julia Method Invalidation Example

When function f is redefined to have a new implementation, function g must also be recompiled to ensure that the result remains accurate even with the redefinition of f.

of backedges. Then, when any of those methods was recompiled (f), the backedge graph would be traversed and the relevant code instances would be invalidated [5].

Method invalidation is a Julia feature that significantly increases compile times, due to the recompilation triggered by invalidation. However, its presence motivates the need for a lower-latency JIT compiler for Julia.

3.3 Precompilation and Code Images

3.3.1 Precompilation

As one solution to the latency problem, Julia developers have developed a precompilation system that caches type-inferred code. Precompilation is a way of saving the results of type inference in a manner that the saved results can be loaded in future sessions. As type inference can take significant amounts of compilation time, carefully chosen precompilation statements can provide large savings for an end-user that adheres to those call signatures [11].

Precompilation is primarily of interest to package authors, as packages are con-

sidered more stable than code written by end-users. When packages are added to a local Julia environment, the Julia runtime will look for and locally cache the results of precompilation statements defined by the package author. Then, users may load the package at any time in the future and benefit from the savings in compilation time [11].

3.3.2 System Images

Another solution that has been developed is known as a system image. Rather than save just the type-inferred code from precompile statements, system images also save the native code that results from compilation of the precompile statements and allow loading of that native code in a future session. A system image therefore has the capability to vastly reduce compilation time from a carefully designed Julia program [15]. Julia also ships with a mode that can eliminate the compiler by loading a system image and restricting programs to only call methods that are defined inside that system image. Furthermore, system images may be compiled for multiple CPU architectures [3]. This multi-CPU compilation is referred to as multiversioning, and is a critical feature enabling usage of higher performance vector instructions on CPUs that support them while maintaining fallback methods for CPUs that do not support vector instructions.

However, system images do have some drawbacks. Unlike standard precompilation, which can be done incrementally for each package, at most one system image can be used at a time by the Julia runtime. Furthermore, adding or removing methods from the system image requires a full recompilation of that system image, which can take significant amounts of time due to the large number of methods contained within one. The system image used by base Julia for its most critical functions contains over 25000 functions and can take time on the order of minutes to finish compiling.

3.3.3 Package Images

Package images are a new feature introduced in Julia 1.9. Rather than relying on the system image framework to cache native code, package images are an extension of the precompilation framework that simply caches native code as well as inference results locally. As a result, package images benefit from the advantages of precompilation over system images, while also further reducing compilation latency from packages. However, like system images, package images do take a large amount of time to precompile when the package is first installed, and this cost will be repeated each time either the package or one of its dependencies is updated.

Package images also compare unfavorably to system images in terms of relocatability. System images may be uploaded to a web server and made available for download by users. By contrast, package images cannot be shared in the same way, as they are dependent on the local filesystem and the order in which packages are loaded. Thus, package images do not serve as a complete replacement for system images.

3.3.4 Method Invalidation

However, all of these solutions face significant challenges with regards to method invalidation, as type inference results and compiled code may be invalidated if methods are redefined. This can occur if two packages define methods with the same name, and both are loaded into the current session. If such invalidation occurs, future calls to the conflicting method and any methods that call into the conflicting method may need to be reinferred and recompiled [12].

Method invalidation also has a secondary impact on package images. Package images are compiled under the assumption of a certain set of method definitions being present. If that set of method definitions changes, the package image may be invalidated and some native code may have to be recompiled, as different methods could be called. Thus, package images have an additional load time cost due to having to check each of its compiled methods to ensure no recompilation is required. System images do not suffer from this problem, as system images are loaded before any other

code and therefore cannot have method conflicts when they are loaded.

3.4 LLVM's New Pass Manager

LLVM has two different pass manager systems, referred to as the legacy pass manager and the new pass manager. The legacy pass manager has existed since the beginning of LLVM, and was the default pass manager until recently. The new pass manager has become the default pass manager, as a result of its design for reduced compile times and more efficient compilation [9].

As background, both pass managers have analysis passes that extract information about the IR and transform passes which change the IR. Each of these passes is then split into module, call graph, function, and loop passes, each of which operates on their respective IR unit. By changing the IR, transform passes may invalidate analyses that were performed earlier, and those analyses may have to be recomputed as a result.

The legacy pass manager interface allowed transform passes to specify upfront which analysis passes were required by that transform pass, and which ones would be preserved. When the pass was run, the result of running that pass would be a boolean indicating if the IR was changed. If the IR was changed, then any pass that was not specifically preserved upfront would be invalidated and need to be recalculated, a process that could be rather computation-intensive over a large amount of IR.

The new pass manager interface provides more granularity and flexibility compared to the legacy pass manager. Instead of declaring dependencies upfront, passes may now demand analyses when necessary, which saves computing analyses if they end up never being used by a pass. Such a situation could occur if the pass makes an early exit due to a precondition not being met. Furthermore, the result of running a transform pass on the IR unit has changed to a set of preserved analyses, which may be all analyses or a subset thereof. This allows passes to preserve certain kinds of analyses conditionally, which can provide further savings in not having to recompute those analyses as would have been required by the legacy pass manager.

3.5 Compiler Concurrency Support

3.5.1 Global Compilation Lock

Julia's compiler was protected by a single compilation lock, which guards all of type inference, LLVM IR generation, and native code generation [2]. This lock simplified implementation of the compiler, but ultimately proves an obstacle to multithreaded compilation. Multithreaded compilation is a useful feature, as it not only enables threads performing unrelated tasks to progress independently of each other, but also is a requirement for moving compilation to background threads.

3.6 JIT Compilation Strategies

3.6.1 HotSpot Java Virtual Machine

The Java programming language is typically run on the HotSpot Java Virtual Machine (HotSpot JVM), which implements a JIT compilation strategy known as tiered compilation. Initially, Java bytecode is interpreted directly by the HotSpot JVM. When a method is found to be called often, the HotSpot JVM compiles a slightly optimized version of the method with profiling instructions added. Then, once sufficient profiling data has been collected, the HotSpot JVM will compile a finalized version of the code, using profiling data to enhance the applied optimizations [1].

The HotSpot JVM also supports speculative optimization and deoptimization of code. Speculative optimization of code refers to performing optimizations that may not always be correct. When the incorrectness is discovered during execution of the optimized program, the compiler will deoptimize the method by returning it to the interpreter to potentially be recompiled with the additional incorrectness information accounted for. This enables more aggressive optimizations to be used initially, without needing to compile extensive code to account for edge or error cases [16].

Currently, the Julia compiler implements none of these optimizations. However, some of the work presented in this paper may be extended in the future to support

these optimizations.

3.6.2 On-Stack Replacement

On-stack replacement is a technique used by some JIT compilers to be able to optimize code concurrently with running an application [8]. Typically, code replacement occurs by replacing a call to an unoptimized function with a call to an optimized functions. This presents a problem when an unoptimized version of a function has already begun execution, but the optimized version becomes available and is significantly faster. On-stack replacement allows switching execution to the optimized code even after execution of unoptimized code has begun.

Currently, Julia does not have support for this feature, and support for on-stack replacement would likely come after tiered compilation support is implemented.

Chapter 4

Image Generation Parallelism

4.1 Overview

As mentioned in sections 3.3.2 and 3.3.3, Julia is capable of compiling machine code ahead of time and using it to avoid recompilation in future runs. For optimal performance of the compiled code, Julia runs its entire optimization pipeline on all of the methods being compiled. However, this step takes a long time, as each system image or package may have hundreds of methods that need compilation [6]. Thus, parallelizing this optimization step provides large speedups to the build time of system images and package images while making optimal use of the available system resources.

4.2 Procedure

4.2.1 Parallelization

LLVM has significant support for concurrency in the form of multiple contexts, as described in 2.3.2. Therefore, the process for parallelizing the image generation, as shown in figure 4-1, is centered around creating fragments of the original module in different LLVM contexts and running them on multiple threads.

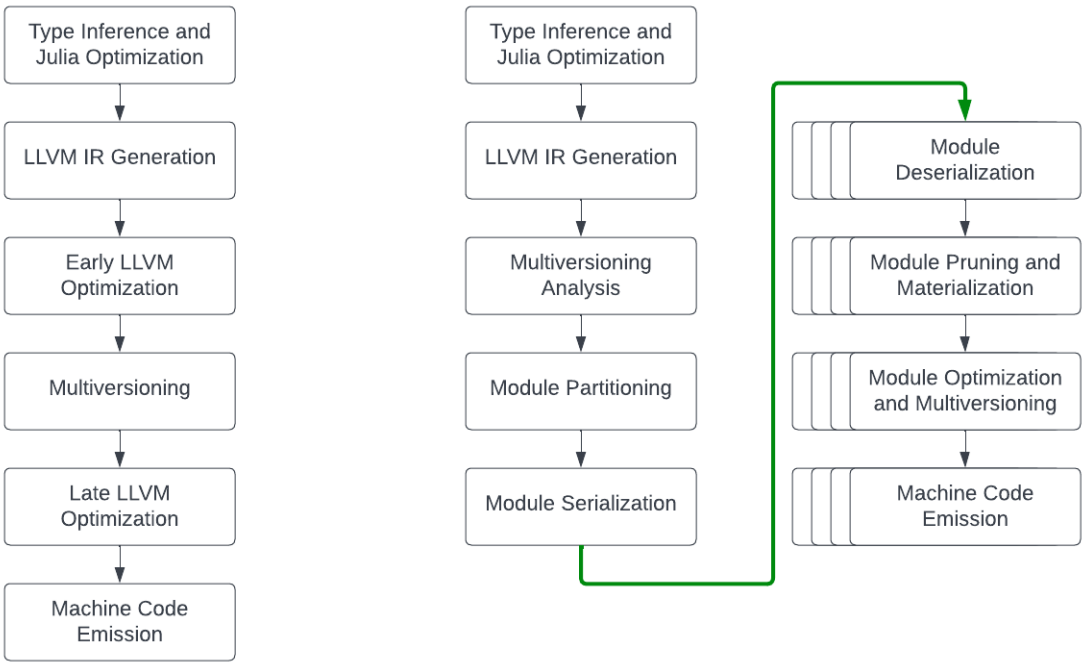

Figure 4-1: Image Optimization Parallelization Process

Original (left) and parallel (right) process for generating Julia code images. Steps that run on multiple threads are shown in the third column on the right and denoted with multiple overlapping boxes. Multiversioning information is analyzed prior to optimization in the parallel generation to allow function cloning and fixups to happen on multiple threads in parallel and avoid a synchronization point in the middle of the optimization pipeline.

4.2.2 Module Partitioning

The process of creating these fragments has a crucial role on the speedup factor of parallelization. The overall runtime of the parallel step of image generation is always the maximum time taken to optimize and emit a fragment. This means that optimal parallelism requires an equal load to be shared by all of the processors. Thus, we implement a partitioning algorithm that estimates the added workload each module feature adds to a module and fragments the module accordingly.

This partitioning step relies on the computation of a module's complexity score, as defined below.

$$S_M = n_{gv} + n_{ga} + \sum_{f=1}^{n_f} S(F_f) \tag{4.1}$$

In equation 4.1, we define S_M as the complexity score of a module, n_{gv} as the number of global variables, n_{ga} as the number of global aliases, and $S(F_f)$ as the complexity score of function F_f. We then define the complexity score of a function F_f as follows.

$$S_F = (1 + n_{bb} + n_i) * n_c \tag{4.2}$$

In equation 4.2, we define S_F as the complexity score of a function, n_{bb} as the number of basic blocks in a function, n_i as the number of LLVM instructions in a function, and n_c as the number of times multiversioning will clone that function.

We then distribute work by sorting the list of global values of the module in descending order of complexity score. Then, we assign each global value in order to the fragment with current lowest complexity score, which gives us a reasonably well partitioning of work.

4.2.3 Serialization and Deserialization

Transferring modules between contexts is hard to do in LLVM, and requires the serialization of the module to bitcode and deserialization from that bitcode into the destination context. However, this serialization and deserialization process is slow. To reduce time spent in this context deserialization, we take advantage of LLVM's

support for lazy bitcode deserialization, which avoids loading function bodies from bitcode until specifically requested. This allows us to serialize the entire module to bitcode once on the main thread, then deserialize the module on each worker thread with lazy deserialization. Once the module is available for operation on each thread, we then delete the functions, global variables, and global aliases that are irrelevant to the thread's assigned fragment workload, and only then force the remaining function bodies to be loaded from bitcode. Running the deserialization and partition pruning on multiple threads increases the parallel fraction of the code without requiring the main thread to do all of the deserialization work.

4.2.4 Multiversioning

One complicating factor of the image generation process is the ability for Julia to generate code optimized for multiple architectures in the same binary. This process, called multiversioning, collects information about every function in the module, clones functions and marks them for optimization with architecture-specific flags, and creates a few large arrays with information about the different functions that have been cloned for the specified architectures. Then, when Julia loads the image, the Julia loader inspects these arrays, selects the functions that are best optimized for the architecture that it is being run on, and links the functions into the current process. Generating these large arrays were an inherent blocker to parallelization of the image generation, as splitting up the module would involve spreading functions out among different fragments. Additionally, the multiversioning pass would no longer have access to all of the functions in the module, which was another blocker to parallelization.

To work around this, the multiversioning pass is first split into two to collect the information in an initial step and to later do its cloning with that information available. Then, we modified the loader to load multiple fragments of the module from the same image. This allows each fragment to be emitted separately, without needing to fuse the arrays from separate fragments back together. Together, this parallelizes of the multiversioning pipeline by first collecting the information necessary for multiversioning, then running the optimization pipeline with a pass that did the

cloning and array emission steps of multiversioning alone, and then emitting each fragment of the original module to the same shared library.

4.3 Performance Metrics

Here, we analyze the parallelization of image generation in two contexts: system image creation, and package images created by import Pkg; Pkg.add("OrdinaryDiffEq"). Data for the system image was collected with 10 rounds of system image building, while data for the package images was collected with 6 rounds of building, with data collected at each thread count from 1 to 64, inclusive. The default system image was built without any multiversioning, while the multiversioned system image was built with `JULIA_CPU_TARGET` set to `"generic;sandybridge,-xsaveopt,clone_all; haswell,-rdrnd,base(1)"`.

4.3.1 System Image

Speedup Analysis

To analyze how building the system image was sped up using parallel image optimization, the system image was built 10 times at thread counts varying from 1 to 64 threads. The time spent in various stages of the build process was measured, then used to derive speedups when using N threads to optimize the system image as compared to using 1 thread to optimize. The results from this experiment are plotted in figure 4-2.

As seen in figure 4-2, parallelization of the optimization step alone results in very large speedups in the total amount of time spent building the system image. For the default system image up to 8X speedups are observed, while for multiversioned system images speedups approaching 16X are observed. This increase in speedup for the multiversioned system image is due to additional work being done in the optimization pipeline, which increases the parallel fraction of the code relative to the default system image and thus enables greater speedups as indicated by Amdahl's

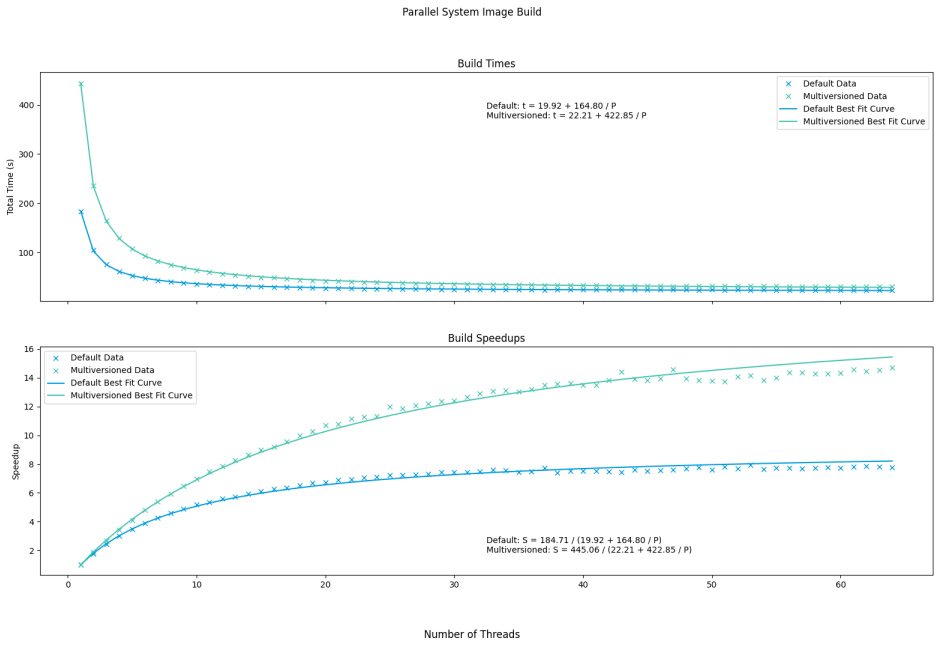

Figure 4-2: System Image Parallelization Speedups

Parallelizing system image optimization results in significant reductions in runtime and speedups ranging up to approximately 8X for a default system image build and 16X for a multiversioned system image build. The best-fit equations suggest a minimum speed of 19.92 seconds for the default system image (9.27X speedup) and 22.21 seconds for the multiversioned system image (20.03X speedup) with infinite cores available.

law.

By curve fitting the equation $t = T_S + T_P/P$ to both the default and multiversioned system image times, where t represents the build time, T_S represents the serial time of the program, T_P represents the parallel time of the program, and P represents the number of processors, we arrive at a figure of $T_S = 19.92$ seconds and $T_P = 164.80$ seconds for the default system image, and $T_S = 22.21$ seconds and $T_P = 422.85$ seconds for the multiversioned system image.

Parallel speedups are typically fairly sensitive to the distribution of work between processors. Thus, an analysis on the partitioning algorithm was conducted along with the speedup study to assess its performance and identify areas for improvement.

Partitioning Analysis

Firstly, the partitioning algorithm relies on the computation of a complexity score, which is used as a proxy for runtime. If the complexity score does not relate to the actual runtime of optimization, then work will not be split up among the threads equally, which increases the span of optimization. Therefore, the complexity score and time to optimize each partition was measured in the system image building process, for thread counts from 1 to 64. Higher thread counts result in each partition having a lower complexity score, which enables analysis of the correlation between complexity score and runtime. The results of this experiment are shown in figure 4-3.

Figure 4-3 shows that for both the multiversioned and default system images, as complexity score increases, the runtime required for optimizing the system image also increases. This indicates that the complexity score heuristic is valid for estimating expected runtime of optimizing an image. However, as the system image is partitioned, the variation in runtime increases with an increasing number of partitions, which is visible in the wider spreads towards lower complexity values in figure 4-3. This indicates that there exist some factors that modify the optimization time of a module that are unaccounted for in the given complexity score formula.

As stated in section 4.2.2, the optimization time in a parallel context is determined by the runtime of the longest-running partition of the module. By using the conclu-

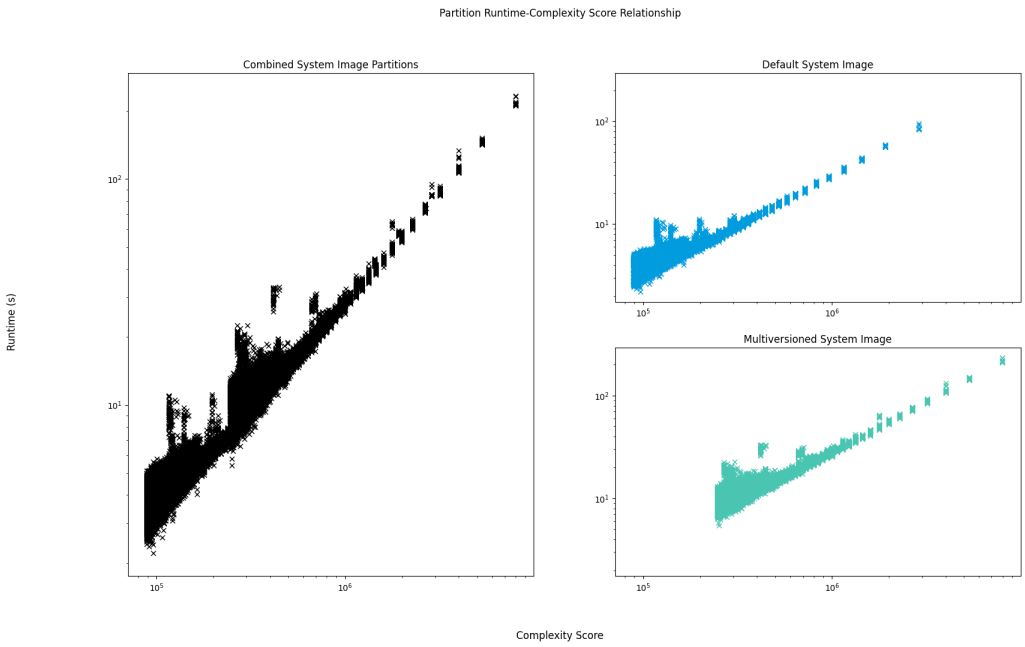

Figure 4-3: System Image Complexity Score Analysis

Complexity scores from partitioning the system image show that at higher complexity scores, partition optimization time is closely linked to complexity score. As complexity score decreases, variation in the complexity score-runtime relationship increases, a relationship which holds across both the default and multiversioned system image. Thread count is inversely related to per-partition complexity and runtime.

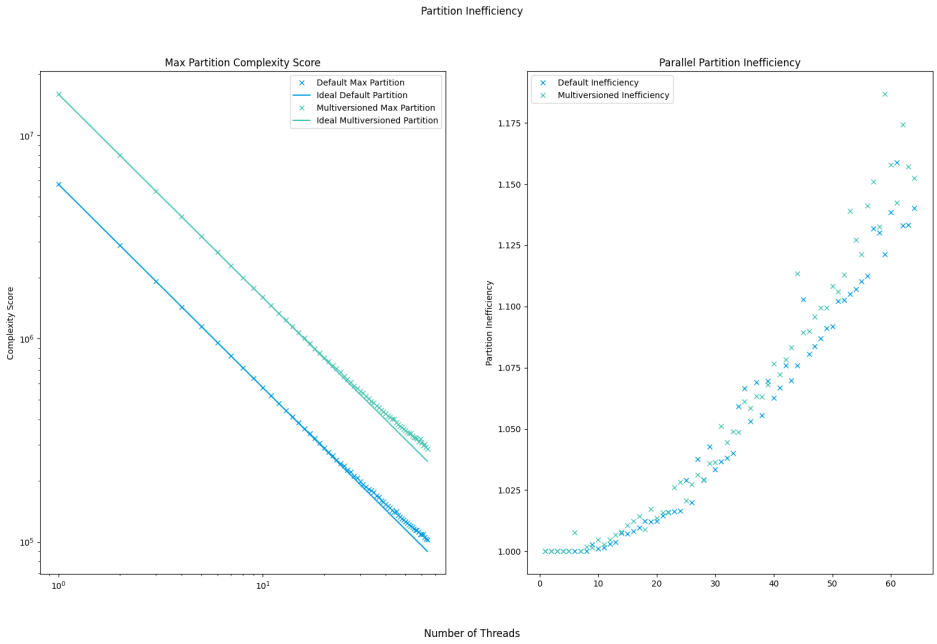

Figure 4-4: System Image Partitioning Analysis

The chosen partitioning algorithm is very effective at partitioning the system image into a small number of partitions (< 10), but shows increased maximum partition complexity scores as the number of partitions grows. However, the maximum partition complexity score remains within 20% of the ideal partition complexity score for all thread counts up to 64 threads.

sion that complexity score is a valid proxy for runtime, the goal of the partitioning algorithm can be converted to partitioning the module into partitions such that the maximum complexity score of any partition is as small as possible.

To assess the effectiveness of the partitioning algorithm, we first note that the ideal partitioning algorithm results in partitions that have complexity scores exactly equal to the total complexity score of the module divided by the number of threads, formalized in equation 4.3.

$$S_{P,I} = \frac{S_M}{P} \tag{4.3}$$

Then, we define an inefficiency metric which determines the degree to which the produced partition exceeds the optimal partition. This is defined as the observed maximum complexity score of any partition when the module is partitioned into P partitions. This is formalized in equation 4.4, where S_P represents the complexity score of a partition, P represents the number of threads, and I represents the inefficiency metric.

$$I = \frac{\max(S_p \forall p \in P)}{S_{P,I}} \tag{4.4}$$

Defining the inefficiency score in this manner allows comparison of different modules in a standardized method. However, the ideal partitions represented by equation 4.3 are not necessarily achievable, as large functions cannot be split among multiple modules and therefore block parallel speedups in that manner.

To confirm the acceptability of the chosen partitioning algorithm, the maximum partition complexity score for every thread count was measured, along with the total weight of the system image. The results of this measurement are shown in figure 4-4, which shows that the maximum partition complexity score remains relatively close to the ideal maximum partition complexity score at low numbers of partitions. As the number of partitions increases, the maximum partition complexity score begins to diverge further and further from the ideal complexity score, achieving a maximum inefficiency score of approximately 17% over the ideal partition. Since the ideal

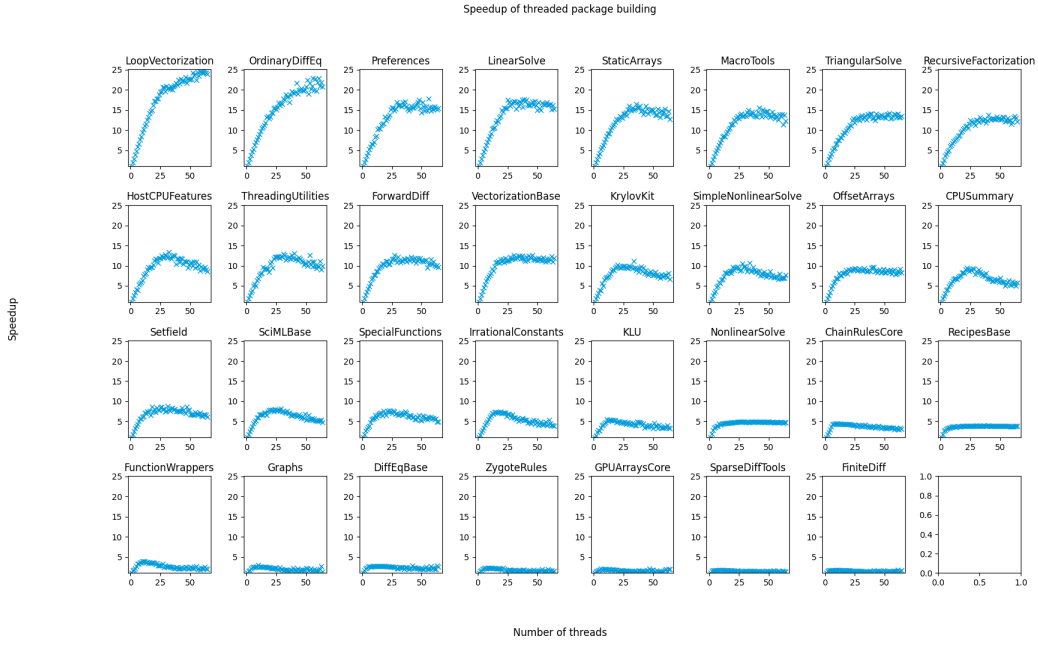

Figure 4-5: Package Image Parallelization Speedups

Parallel speedups from various packages as a result of precompilation triggered from adding the OrdinaryDiffEq package. Packages are sorted from left to right, top to bottom, in decreasing order of maximum speedup from parallelization.

complexity score is a lower bound, and 17% is not a very high inefficiency score, we can conclude the partitioning algorithm is effective at distributing work among the available computational resources.

4.3.2 Package Images

Package image data was collected by adding the package OrdinaryDiffEq to the local environment with the JULIA_NUM_PRECOMPILE_TASKS environment variable set to 1 and iterating on the number of image parallel threads sequentially. Of the 112 packages and dependencies that were precompiled as a result, 81 packages fell below the minimum threshold of 1000 complexity score to be assessed for parallelization, and 31 packages were optimized on multiple threads.

The speedups for the parallel packages when building images with multiple threads is visible in figure 4-5. Each of these packages see significant speedups when compiling with multiple threads, although packages that see smaller speedups also see a

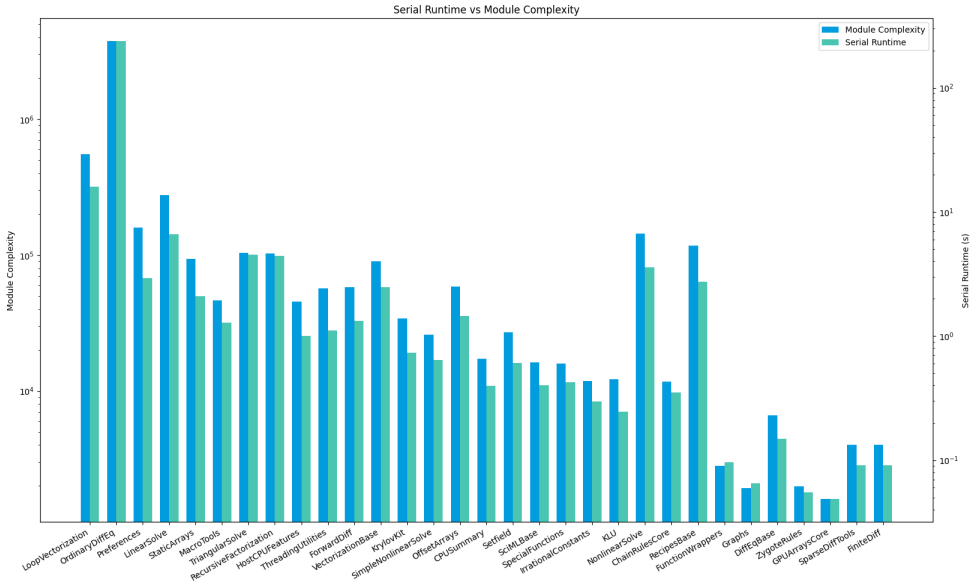

Figure 4-6: Package Image Complexity Score Analysis

Serial runtime and complexity scores for parallel-capable modules. The modules are sorted in decreasing order of maximum parallel speedup from left to right, which shows that generally maximum parallel speedup increases as both a function of module complexity score as well as serial runtime. Thus, module complexity serves as a reasonable proxy for serial runtime for packages, and qualitatively corresponds to the expected parallel speedup. The exceptions of NonlinearSolve and RecipesBase are discussed in the text.

decrease in speedup when thread counts rise beyond a package-specific optimum. To analyze why particular packages saw greater speedups than others, the initial module complexity and runtime with one thread were compared in figure 4-6, where package names are sorted in decreasing order of maximum parallel speedup from left to right. Overall, complexity score and serial runtime track each other well, while qualitatively there appears to be a relationship between these two factors and maximum parallel speedup.

However, there are a pair of outliers in the form of NonlinearSolve and RecipesBase, which show high complexity score and serial runtime but relatively low parallel speedup. This may be explained by the presence of a few large functions in these packages, which cannot be effectively distributed among multiple threads in the same way that many small functions can be. Evidence to support this theory comes from

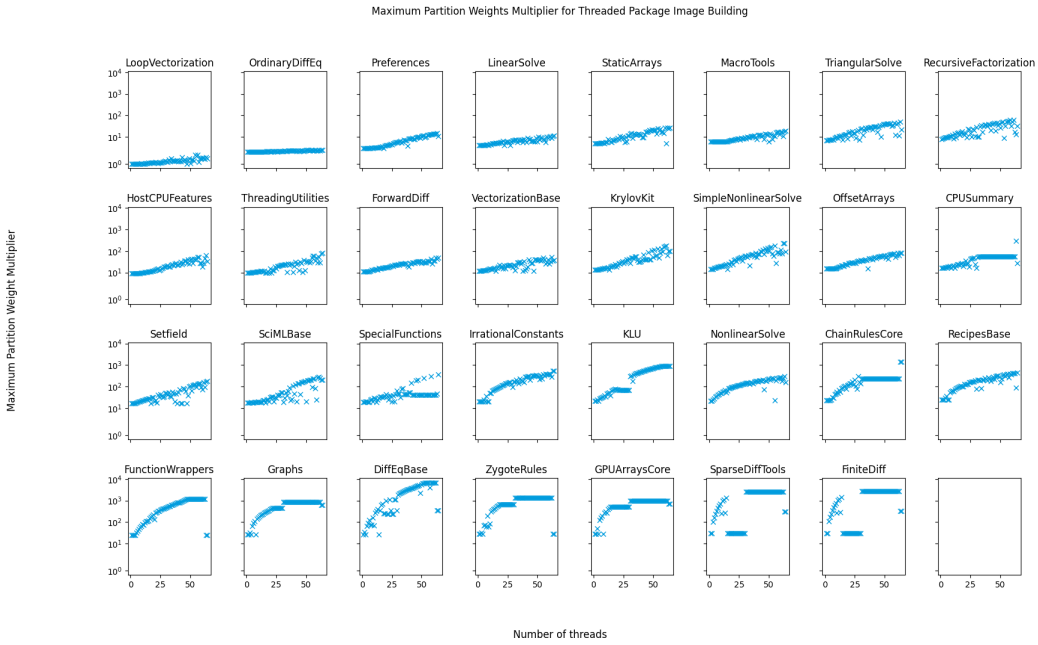

Figure 4-7: Package Image Partitioning Analysis

Multiplier of maximal partition by complexity score at varying thread counts for different packages. Higher multipliers indicate suboptimal partitioning, while smaller multipliers indicate more optimal partitioning.

figure 4-5, which shows that parallel speedup peaks at a small number of threads but does not decrease at high numbers of threads, the way that other packages with similar low parallel speedups do. This indicates that there exists partitions comprised primarily of a single large function that cannot be split, which then occupies the same amount of time to optimize and emit even when the number of threads is increased. Further evidence to support this suboptimal partitioning theory is provided in figure 4-7, which shows high maximal partition multipliers for both of these packages that match packages with similar parallel speedup patterns, indicating that a few large unsplittable partitions are contributing the majority of the runtime.

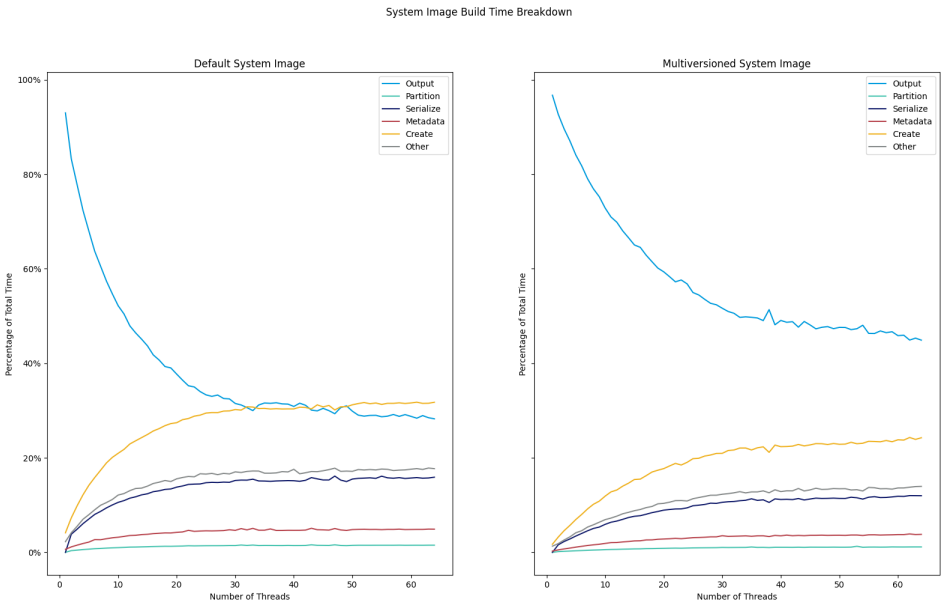

Figure 4-8: Parallel System Image Build Time Breakdown

Increasing thread count causes module creation and serialization to take up significantly larger proportions of the image creation time. For the default system image, module creation time surpasses module output time, while a considerable margin between output time and module creation time still exists for the multiversioned system image.

4.4 Further Optimizations

4.4.1 Optimal Partitioning

While the current parallelization work has resulted in significant speedups in both system image build time and package image build time, there are still potential extra speedups that could be obtained from either more or better parallelization. As shown in figure 4-8, the fraction of time spent partitioning the module is much lower than the fraction of time spent optimizing the module. This indicates that a different partitioning algorithm may provide a net speedup in the overall time of the build process, trading some extra time spent partitioning the image for a more balanced parallel workload. Alternatively, the complexity score used by the partitioning algorithm could be further investigated to include the additional component causing runtime variation noted in section 4.2.2. Refining the complexity score would also improve parallel work distribution by giving the partitioner a clearer view of which global values take the most work to optimize, and thus reducing additional work present in their partitions.

4.4.2 Multi-Module Generation

Another method of reducing the system image build time is to alter the method by which the system image LLVM module is created. Currently, the build process compiles an array of methods into an array of modules, and then fuses them into one module before optimizing it. Now that the optimizer is partitioning the one module into multiple, it may make sense to pre-partition the array of methods and generate LLVM IR from those partitions in different contexts directly. Doing so enables the method-level partitions to be parallelized, and also removes the serialization step as the different method partitions would already be in different LLVM contexts. Since figure 4-8 suggests that module creation and serialization occupy the majority of the non-optimization time, optimizing and/or parallelizing these steps would likely result in the biggest speedups.

4.4.3 Saving JIT-Compiled Code

Due to Julia's dynamism, precompilation occurs by executing precompile statements, tracking which methods are called, then compiling them in a reloadable format and saving them to disk. During the execution process, most of the methods that are called end up being compiled by the JIT compiler. This compilation work is however wasted, as the JIT compiler does not compile code in a manner that enables the code to be reloaded after being written to disk. Altering the JIT compiler to produce this kind of code would remove most of the methods from having to be compiled twice, resulting in potentially large savings if no multiversioning is required. If multiversioning is required, however, the compile twice approach may be faster due to the available parallelism that cannot be introduced at the JIT compiler stage.

Chapter 5

JIT Concurrency

5.1 Overview

As chapter 4 covers speedups gained from parallelizing ahead-of-time (AOT) compilation, this chapter covers speedups gained from concurrent just-in-time (JIT) compilation of methods. The primary difference between the two strategies from a performance perspective is that while parallelization performs the workload distribution itself, concurrent JIT compilation simply permits multiple application threads to be running compilation simultaneously. In this context, we hoped to see gains from applications that exposed a large amount of work to the compiler on multiple threads.

Here, we consider only concurrency in the type inference phase of the compiler. Concurrency in the codegen phase poses a few extra challenges that currently make it infeasible to deploy, which are discussed in section 5.4.4.

5.2 Procedure

5.2.1 Lock Removal

The formal procedure of removing the global compilation lock involves progressively shrinking the region covered by the lock until no code is executed within the locked region, at which point the lock and unlock calls may simply be deleted. This process

of identifying code that used shared resources, either eliminating or correcting those uses to be thread safe, and then shrinking the lock around those uses was iterated on several times to remove the lock around type inference. As an example, we consider how saving type-inferred method data was made to be thread safe.

5.2.2 Atomics

Many of the data structures that type inference uses are updated by using atomic primitives that are safe to execute from multiple threads. By relying on the ordering and visibility of the effects of these atomic primitives, the data structures do not require the use of a lock to guard against concurrent modification. This also allows increased concurrency as no thread must wait for a lock to be acquired before performing type inference.

In the context of concurrent type inference, the type-inferred method data was stored to a field in the code instance non-atomically. By marking the stores and loads as atomic, and verifying that the relevant operations would observe the code instance in a consistent state given the existence of a particular value for that type-inferred method data, accesses to that field were made thread safe and thus enabled the global compilation lock to be removed around type inference.

5.3 Performance Metrics

5.3.1 Synthetic Benchmark

Since the main speedups were to be expected from applications that exposed work to the compiler on multiple threads, a synthetic benchmark was designed with the intent of measuring the performance gap. The design goal of this benchmark was to allow varying the complexity of type inference, as well as permitting repeated inference at the same complexity level. This was achieved by creating a tree of recursive function calls, with the total number of nodes being equal to the product of $p1$ and $p2$. The ability to repeat inference at the same complexity level was accomplished by varying

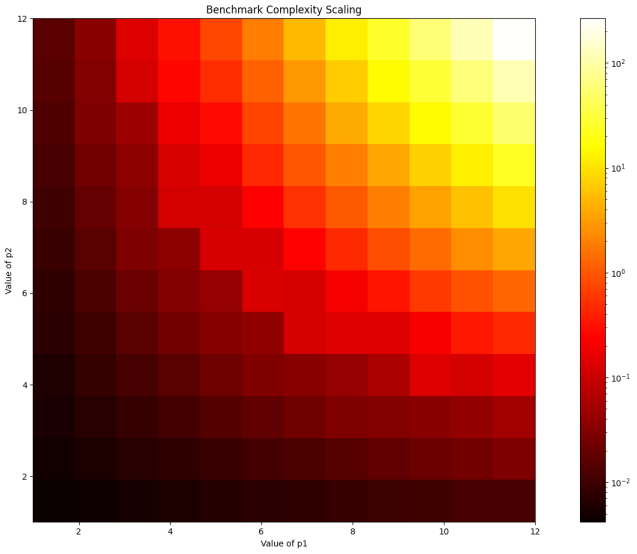

Figure 5-1: Scaling Measurements for Synthetic Benchmark

Plot of median runtimes against different combinations of $p1$ and $p2$, log scaled along the median runtimes in the color axis. An optimal runtime for further measurements was found in the boundary between the red and yellow regions, corresponding to runtimes in the 1-10 second range.

the type of the first parameter. Since Julia triggers type inference whenever a new combination of types is passed to a method, and the first parameter does not affect the inference complexity of the function, this satisfies the stated design goal.

Selection of an appropriate value for $p1$ and $p2$ for further experimentation is critical; too small results in type inference not spending much time in contention, while too large represents a waste of computational time and memory waiting for shared resources. Therefore, we measured inference runtime for a single compilation of the benchmarking function for every pair of numbers from 1 to 12 for both $p1$ and $p2$. The results of this experiment are shown in figure 5-1, which shows the expected multiplicative increase in runtimes as $p1$ and $p2$ were increased. We find that the values of $p1 = p2 = 9$ has an inference time of approximately 4 seconds, which represents an acceptable trade off where contention between multiple threads is easily observable.

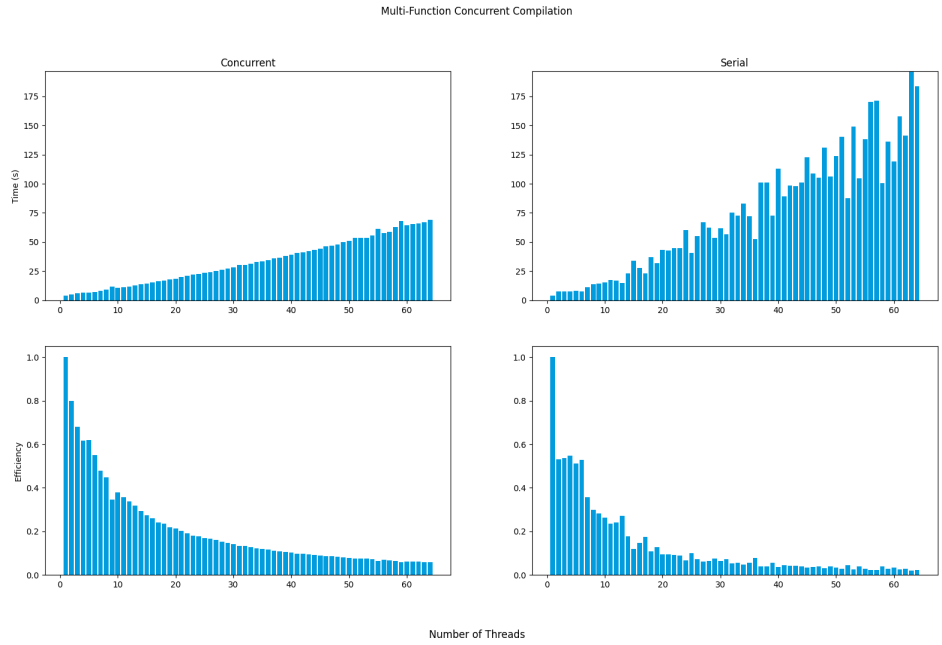

Figure 5-2: Runtimes and Efficiencies of Serial and Concurrent Type Inference for Multiple Different Functions

The runtime of concurrent type inference significantly decreases relative to the runtime of serial type inference with the global compilation lock still held for type inference. Values presented are the median of 10 trials at each thread count. Efficiency scores were computed by dividing the median runtime of a single thread by the median runtime at a particular thread count.

5.3.2 Multi-Function Efficiency Comparison

To compare the effective speedup of concurrent type inference over serial type inference, we conducted an experiment where multiple threads would simultaneously attempt to compile different functions, resulting in contention over shared compilation resources. In serial type inference, the expectation is that threads sequentially acquire the compilation lock and release it one thread at a time, resulting in a workload tied heavily to the number of threads. In the fully concurrent case, we expect zero contention with every method being compiled simultaneously. As figure 5-2 shows, however, the workload does grow as a function of the number of threads even with concurrent type inference; however it grows at a significantly slower rate compared to serial type inference. Furthermore by calculating the efficiency scores, it is clear that concurrent type inference has superior efficiency characteristics at any number

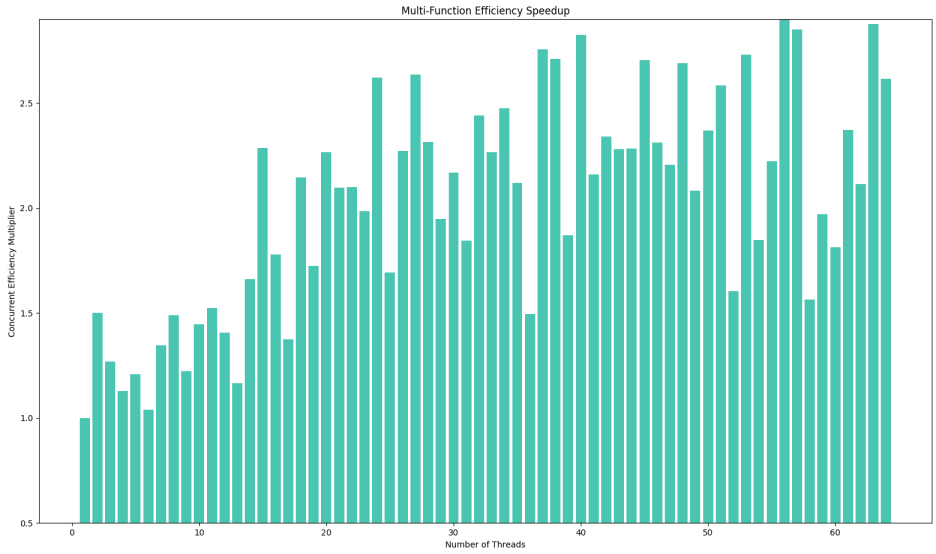

Figure 5-3: Efficiency Speedup for Concurrent Compilation of Multiple Different Functions

Concurrent type inference always presents a net increase in efficiency multiplier at any thread count, maxing out at an approximately 2.5X-3X increase in efficiency multiplier above 32 threads. The efficiency multiplier was computed as the efficiency of concurrent type inference divided by the efficiency of serial type inference for each thread count. Noise in the serial runtime measurements is carried over into the efficiency comparison.

of threads.

Figure 5-3 analyzes the relationship between the efficiency ratios at different thread counts in greater detail. In general, there appears to be a positive relationship between concurrent type inference and greater efficiency scores, with ratios approaching 3X at 64 threads. However, analysis of the data is complicated by the noise in the serial type inference measurements, which causes large variations in the efficiency score ratios. This may be due to the compiler's use of spin locks, which is discussed further in section 5.4.1.

The efficiency gain of concurrent type inference over serial type inference appears to maximize around a ratio of 3, regardless of the number of threads available. This indicates that additional synchronization between threads is reducing compilation throughput.

5.3.3 Same-Function Efficiency Comparison

Since Julia will compile methods for every new combination of types passed into a method, a speedup was also expected in the case where multiple threads attempted to compile the same function for different arguments. This scenario could occur in a real-life workload when a user splits up a list of objects with different types over multiple threads, and then calls a function with each element of that list.

In figure 5-4, it is clear that the timing difference between concurrent and serial type inference when the same function is being inferred with different arguments is significantly smaller than the timing difference when the functions are completely different. Likewise, the efficiency score patterns do not show an obvious clear winner between concurrent and serial type inference. In figure 5-5, it is clear that at low thread counts serial type inference runs in a shorter amount of time than concurrent type inference, though at higher thread counts concurrent type inference regains its advantage.

To understand these more surprising results, we obtained a profiler report, which identified a bottleneck in decompression of Julia IR immediately before inferring the function. This decompression was performed under a lock, which inherently serialized

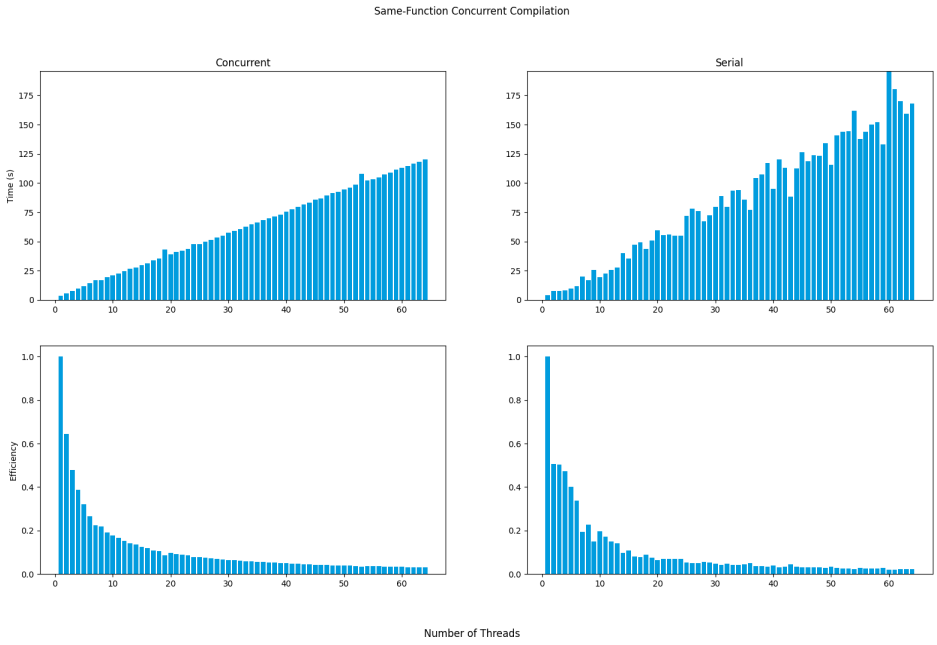

Figure 5-4: Runtimes and Efficiencies of Serial and Concurrent Type Inference for the Same Function with Different Arguments

The runtime of concurrent type inference and serial type inference measure out to be quite similar, though the serial type inference timings are much noisier than the concurrent type inference timings. Values presented are the medians of 10 trials.

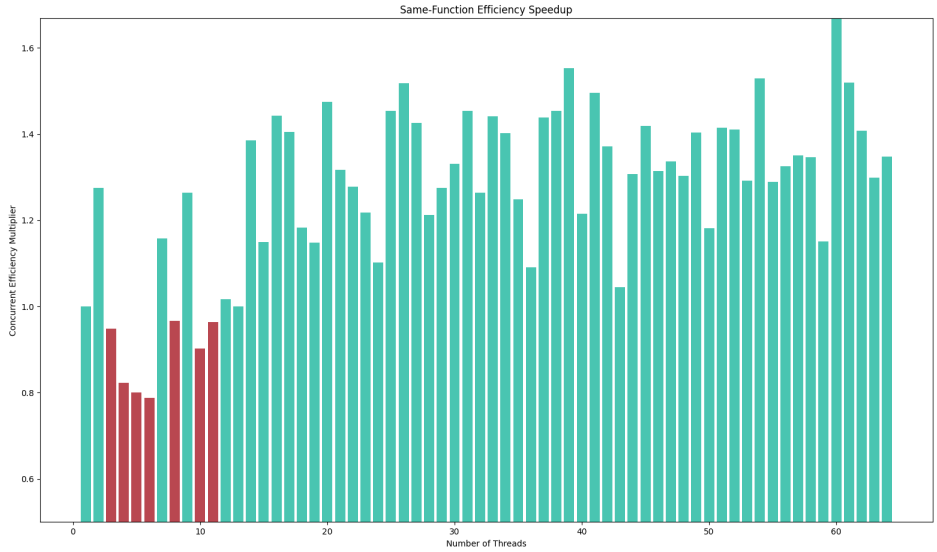

Figure 5-5: Efficiency Speedup for Concurrent Compilation of the Same Function with Different Arguments

At higher thread counts concurrent type inference performs better than serial type inference for multiple thread counts, though at low thread counts it performs worse. Green bars indicate concurrency efficiency multiplier greater than 1, while red bars indicate a multiplier less than 1.

the task between all of the threads. Since multiple different functions do not share the same compressed Julia IR, this bottleneck did not appear in the benchmarks from section 5.3.2 as each thread could decompress in parallel.

The fact that efficiency decreases at lower thread counts compared to serial type inference can likely be attributed to sharing of type inference results from one thread to another. In concurrent type inference, each thread that starts type inference will complete its own inference without results from other threads that may also be inferring similar methods. In serial type inference, however, these results are available to the inferring thread, which may use the results to reduce the time spent in inference. At higher thread counts, the fact that all threads are able to proceed with inference simultaneously outweighs the speed boost from reusing results, which makes concurrent type inference perform better than serial type inference.

5.4 Further Optimizations

5.4.1 Sleeping Locks

Currently, the Julia compiler's internal locks are all spin locks, which are optimized for uncontended locks and unlocks. In the contended case, spin locks will burn CPU cycles in the hope that the lock becomes available soon. Future work could change some of these spin locks to OS-supported sleeping locks, which would reduce power consumption and resource usage in the contended case [14].

5.4.2 Read-Write Lock

As noted in section 5.3.3, the major bottleneck experienced by inferring the same function with different arguments ends up being the decompression of Julia IR, which is guarded by a simple lock. Since decompression is inherently a read-only operation, decompression of the IR could be guarded by a read-write lock [7], which permits multiple readers to access data at the same time. In the presence of a read-write lock for IR decompression, results more closely matching the multiple different methods performance improvements would be expected.

5.4.3 Shared Type-Inference Workqueue

Another method of increasing performance from concurrent type inference is to reuse inference results more aggressively. Currently, type inference results may only be reused when a thread finishes type inference. If thread A starts type inference, and then thread B needs the results of type-inferring the same method, thread B will begin type-inferring the method even though thread A might finish soon. With a more sophisticated work-tracking algorithm, type inference results could be dispatched once, with dependent tasks waiting for the results to be computed. This would save on CPU resources and potentially some real time, as redundant type inference would never occur and type inference results would be computed as soon as feasibly possible.

5.4.4 Codegen Concurrency

Concurrency in the codegen phase of the compiler is more complicated than the type inference phase. While type inference may run multiple times on the same method, codegen interacts with lower level data structures that are harder to access safely from multiple threads. In particular, LLVM's old RuntimeDyld linker could not handle multithreaded linking, while LLVM's new JITLink linker is not supported on some platforms that Julia supports, which precludes a redesign of parallel codegen. However, JITLink support is getting better, which could permit such a redesign in the future.

Chapter 6

Lazy Compilation

6.1 Overview

Julia compiles methods once for a given set of passed-in argument types. However, since the number of types is unbounded in a program, Julia performs this compilation the first time a function is called with a particular set of argument types; that is, `compile(f, typeof(args)...)` will only ever be called once for the same arguments to `compile`. However, in the process of compiling one method, other methods called by the currently compiling method may also be compiled if their argument types are known through type inference. The goal with lazier compilation here was to delay the compilation of these called methods and thus reduce their impact on compile times.

6.2 Laziness Procedure

Implementation of lazier compilation primarily involved adding LLVM's CompileOnDemandLayer to the JIT compiler, which would then implement the necessary functionality required for lazier compilation. The CompileOnDemandLayer required the use of the new JITLink linker, as well as removing an optimization where LLVM contexts were reused for multiple JIT compilations. Additionally, CompileOnDemandLayer comes with an optional partitioning function, which allows users to control the degree to which laziness occurs. Here, we test the case where each function is individ-

ually compiled when called, and the case where each module with potentially multiple functions is compiled when any of its functions are called. This feature allows the user to trade off between adding additional overhead from splitting out functions into different lazy units and avoiding compiling as much code as possible.

6.3 Performance Metrics

Performance metrics here were acquired by building Julia from source, which involves a number of phases of executing Julia code. By default, building Julia from source will perform these executions at optimization level O0, but by performing the executions at optimization level O3 the different executions act as proxies for different kinds of Julia workloads. These executions of Julia code to build Julia from source are henceforth referred to as bootstrap compilation, as they are necessary to set Julia up for running user code.

6.3.1 Potential Gains

Adding lazier compilation inherently involves some overhead in both the first invocation of the function as well as future invocations, as the CompileOnDemandLayer's laziness is implemented via a function trampoline. Therefore, the call count of all the methods that were compiled to the JIT compiler was recorded, as well as the number of basic blocks and instructions in those methods. The results of which methods needed to be compiled compared to which methods were actually called are shown in figure 6-1.

Figure 6-1 shows that regardless of the degree of lazy compilation used, there exist large savings to be gained during the sys.ji and sys-o.a phases of compilation. These phases compile quite a bit of code that is ultimately never actually called by the program. Furthermore, the plot on the right shows that while using per-module lazy compilation, the number of functions lazily compiled drops significantly, the number of actually compiled basic blocks and instructions shows a much smaller reduction.

This disparity between function count and basic block/instruction count is likely

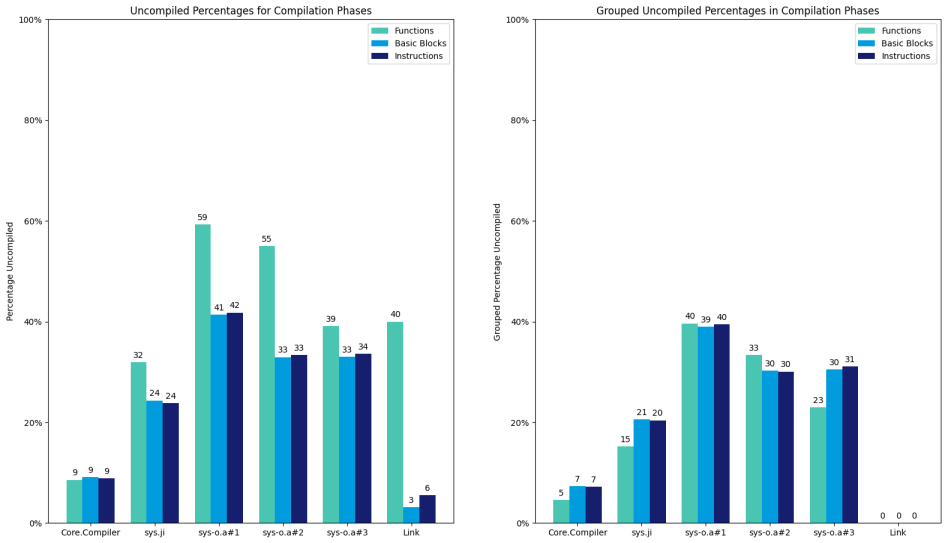

Figure 6-1: Uncompiled Method Fractions in Various Phases of Julia Bootstrap Compilation

Large fractions of code may be left uncompiled during the sys.ji and sys-o.a phases of compilation. The left plot shows the fraction of code that would be left uncompiled when using maximal laziness (i.e. per-function lazy compilation). The plot on the right shows the fraction of code that would be left uncompiled when performing lazy compilation on a per-module level.

due to the existence of Julia ABI stubs, which allow the Julia interpreter to call into compiled code with a fixed interface. These stubs are compiled in the same module as their compiled function, but are infrequently called and typically represent a very small fraction of the complexity and compile time of the module.

The Core.Compiler phase in bootstrap compilation, however, shows a much smaller degree of wasted function compilation. Therefore, this represents a workload which is targeted toward maximal use of compiled functions. Under such a scenario, the overhead from the machinery of lazy compilation is likely to harm performance more than it helps, as all of the code ultimately does end up being compiled. However, the underlying reason behind why Core.Compiler has most of its code is compiled is likely due to poor inferrability. Much of the compiler relies on Julia's dynamism, which will result in fewer static linkages between functions and therefore minimal wasted compilation. In contrast, user code typically places an emphasis on maintaining inferrable code and maximizing type information, which results in additional static compilation whose impact can be mitigated by lazier compilation.

6.3.2 Bootstrap Compilation Laziness

To inspect the true performance impact of lazier compilation, the time to execute various phases of Julia's bootstrap compilation was measured under different build conditions. The results of these measurements are shown in figure 6-2. Firstly, in a change from figure 6-1, the sys-o.a phases were merged into the same timer, and the Link phase was dropped due to the very minor amount of time and functions spent in that phase. Additionally, the runtime of different conditions of implementing lazier compilation was measured, to identify where the biggest performance benefits and overhead arose from.

Figure 6-2 shows that the overhead of per-function lazy compilation far outweighs any benefit that is obtained from lazier compilation, regardless of compilation phase. However, per-module lazy compilation shows a much better performance picture, with better speedups in the sys.ji and sys-o.a phases, as expected from the uncalled function counts in figure 6-1. These speedups range from 2 seconds (4%) in sys.ji to 8

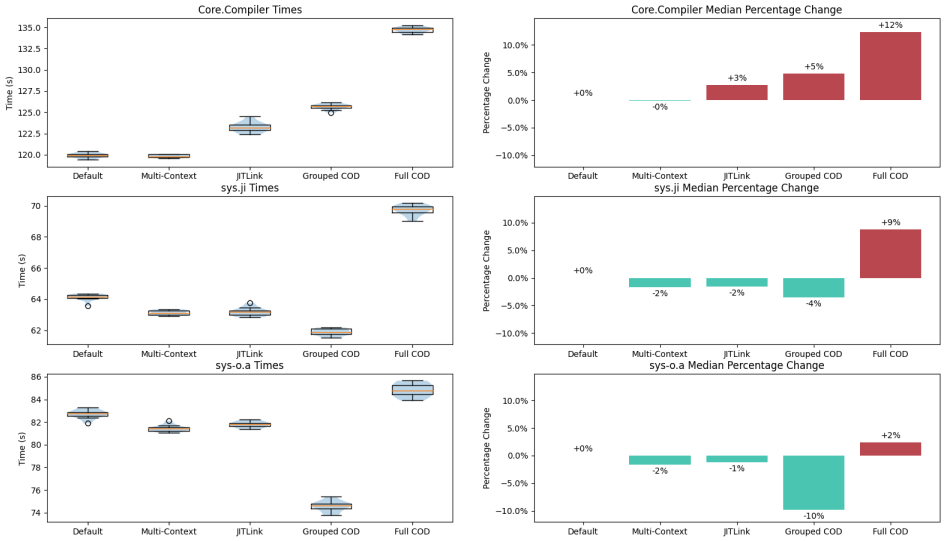

Figure 6-2: Laziness Performance Improvements for Bootstrap Compilation

Per-module lazy compilation sees the largest benefits from lazier compilation, while increased overhead from per-function lazy compilation outweighs any speedups from lazier compilation.

seconds (10%) in the sys-o.a phase. However, it too presents a performance regression in Core.Compiler (5%) due to the almost negligible benefits from lazier compilation, although the overhead is much better than the per-function lazy compilation results.

Additionally, in the Core.Compiler phase the use of the JITLink linker instead of the RuntimeDyld linker normally used accounts for approximately 60% of the performance degradation. This suggests that the laziness overhead in Core.Compiler is much smaller than the absolute difference from the current eager compilation mode, and is related in part to the change of linker. Further improvements to the linker's performance could assist in reducing the linker penalty, and thus make the overhead from lazier compilation more palatable even in the case of poorly inferred code.

Switching from pooled LLVM contexts to a new LLVM context for each compilation appears to have had beneficial impacts on compilation time in the sys.ji and sys-o.a phases of bootstrap compilation. This result was unexpected, as reuse of contexts would typically avoid memory allocation and deallocation overhead, as well as

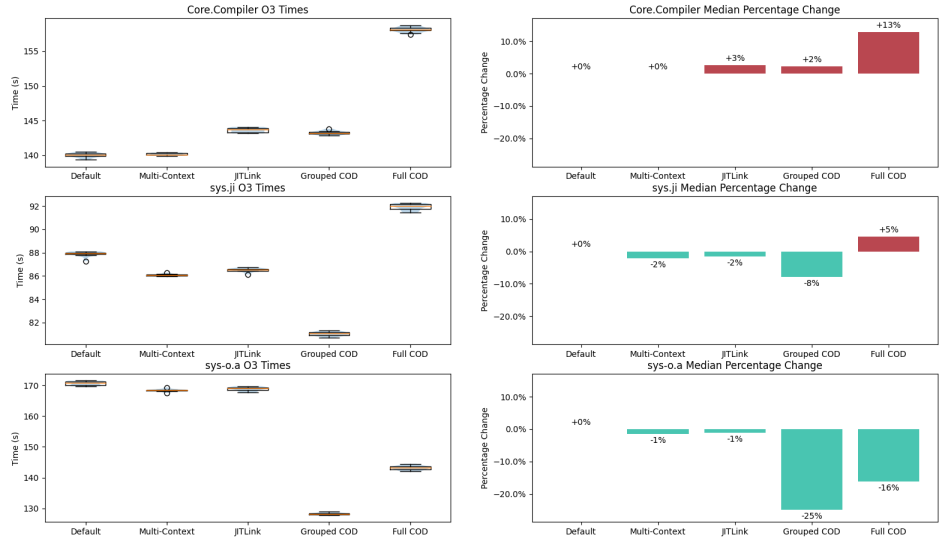

Figure 6-3: Laziness Performance Improvements for Bootstrap Compilation at O3

With higher optimization costs, lazier compilation shows larger time savings, though they do not completely outweigh the increase in compilation time. Compared with figure 6-2, the benefits of lazier compilation are increased, to the point that even the increased overhead of per-function compilation outweighs the added overhead in the case of sys-o.a. Additionally, per-module lazy compilation improves performance over the JITLink-only in the case of Core.Compiler.

permitting reuse of resources between different compilations.

6.3.3 Bootstrap Compilation at O3

Bootstrap compilation occurs at optimization level O0, since the JIT compiled code will not be saved and compilation time takes up a large fraction of the runtime of bootstrapping. However, user code is run at optimization level O3, which entails a longer pass pipeline and more time spent in compilation. Therefore, we repeat the same experiments as section 6.3.2, except that all compilation takes place at optimization level O3. The results of these experiments are shown in figure 6-3.

At optimization level O3, an increased amount of time is spent compiling and optimizing code in the hope of increased runtime performance and overall minimized cost. However, this effort is wasted if the code is never run in the first place. Therefore

we expect to see increased relative benefits from lazier compilation as compared to O0, and this is indeed borne out in figure 6-3. Here, sys-o.a shows an impressive 25% reduction in runtime for per-module lazy compilation, approaching the maximum expected benefit from the counts of uncalled functions in figure 6-1. This indicates that compilation of methods takes up almost all of the time in this phase, and also shows that per-module lazy compilation has very little overhead. In the Core.Compiler phase, per-module lazy compilation even starts to claw back some of the performance regression from switching to the JITLink linker, despite the fact that there were very few functions that would be affected by lazier compilation.

Per-function lazy compilation does better in sys.ji and sys-o.a as well, as would be expected since any reduction in amount of compilation translates to large compile time savings at O3. There does not appear to have been a large change in percent change for Core.Compiler, likely due to the lower impact of lazier compilation there.

6.4 Further Optimizations

6.4.1 Increased Optimization Pipeline

Although significant benefits are not immediately observed from the added LLVM-level laziness, additional benefits are likely to be seen if the pass pipeline becomes more complicated. Since the variable cost of running the pass pipeline on the compiled methods is eliminated until the method is actually called, with total elimination for methods that are never called, this makes the effective time taken by optimization grow slower than without any lazy compilation. Therefore this work could enable a more powerful pass pipeline to be used by default in Julia.

6.4.2 Tiered Compilation

Lazy compilation is fairly close to tiered compilation, as described in section 3.6, where functions are initially interpreted or compiled at a low optimization level but subsequently compiled offline at a higher optimization level. This approach enables

high performance of frequently called methods, while avoiding excess optimization and compilation on methods that are only ever called a few times. Thus, a natural next step for this line of work is the implementation of a tiered compilation system for Julia, which could yield increased benefits in both compile time and generated code performance when section 6.4.1 is considered.

6.4.3 Correctness and Feature Completeness

Currently, the CompileOnDemandLayer implementation of lazier compilation does not pass tests. In particular, one issue is that the implementation does not handle passing arguments to functions in vectors correctly, due to not saving the vector registers when calling back into the compiler. On the Julia side, other features remain incomplete, such as compiling the relevant code without calling into the function. Thus, some additional work is required before the CompileOnDemandLayer can be turned on for the majority of platforms Julia supports.

Chapter 7

New Pass Manager Integration

7.1 Overview

As mentioned in section 3.4, LLVM's new pass manager is supposed to provide improvements in compilation time via a few mechanisms. Here, we analyze its speedups in the few phases of the Julia compiler.

7.2 Performance Metrics

7.2.1 Bootstrap JIT Compilation

To measure the performance impact of the new pass manager, the amount of time spent in the optimization phase of JIT compilation was recorded for both the new pass manager and the legacy pass manager. Furthermore, data was collected for the cases where bootstrap compilation ran at optimization level O0 and at optimization level O3. The results of this experiment are shown in figure 7-1 and figure 7-2.

Figure 7-1 shows that at O0, using the new pass manager actually increased time spent optimizing code, while at O3 using the new pass manager decreased time spent optimizing code. This relation holds constant among all of the phases of bootstrap compilation, indicating that the new pass manager is able to provide performance improvements on Julia's longer and more complicated pipelines rather than its shorter

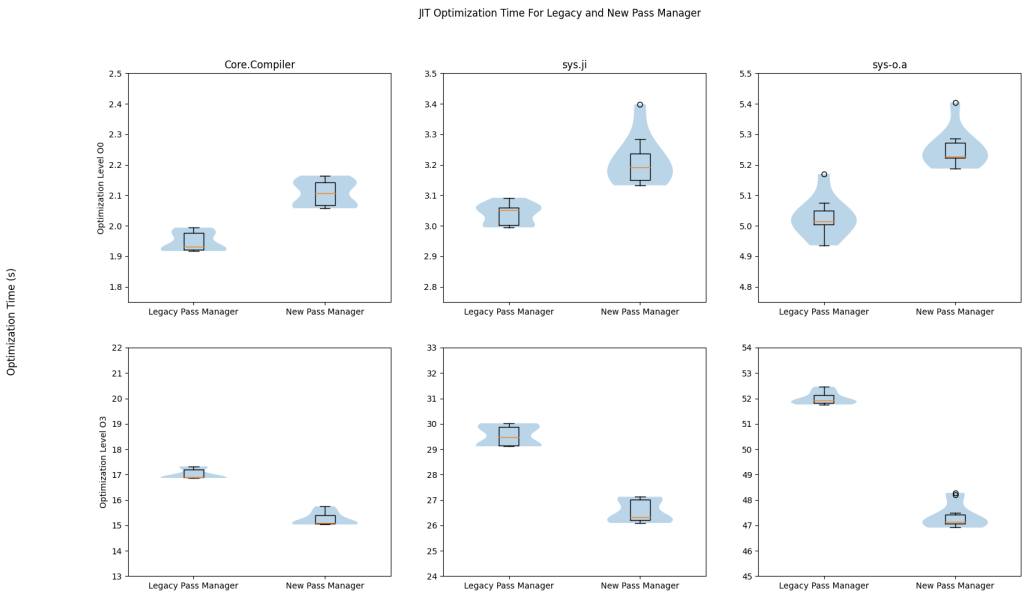

Figure 7-1: Pass Manager Timing Comparison for Bootstrap JIT Compilation

LLVM's new pass manager results in decreased time spent optimizing code in the JIT at optimization level O3 compared to the legacy pass manager, but increased time optimizing code at optimization level O0. Baseline values of each subplot differ to enhance visibility of timing differences. Within the O0 and O3 rows, overall scale is held constant between plots.

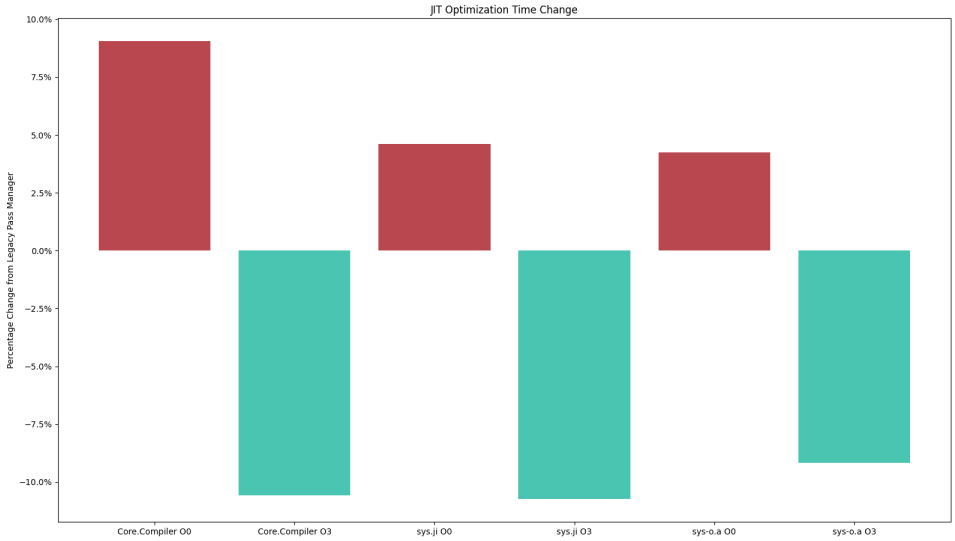

Figure 7-2: Pass Manager Performance Comparison for Bootstrap JIT Compilation

For bootstrap compilation, at O0 the new pass manager has a 5-10% performance penalty compared to the legacy pass manager, while at O3 the new pass manager sees an across-the-board performance improvement of 10%.

and simple ones. Additionally, figure 7-2 provides more numerical insight into these timing measurements, showing that at O3 the new pass manager reports speedups of 10% across all phases. At O0 the penalty is more varied, with Core.Compiler observing a 10% slowdown, but sys.ji and sys-o.a both showing only a 5% slowdown. In raw times, O0 runs much faster than O3, and the savings from the new pass manager are unable to make up for that difference in timings.

The reason why the O0 pipeline sees a slowdown, while the O3 pipeline sees a speedup from the new pass manager likely comes down to the way the pipelines are structured. The O3 pipeline attempts to optimize code to squeeze out as much performance as possible, which involves inserting many LLVM-provided passes into the pipeline. These LLVM-provided passes rely more heavily on IR analyses that LLVM also provides. With the new pass manager, there is better infrastructure for reporting when analyses need to be recomputed and incrementally updating analyses instead of recomputing them from scratch. Since LLVM itself made the transition

to use the new pass manager by default, those LLVM-provided passes and analyses have been made aware of each other and thus take full advantage of the new pass manager's functionality.

By contrast, the O0 pipeline includes very few LLVM passes, and mostly relies on passes written by the Julia developers with compatibility shims to hook into the new pass manager's interface. These passes have not been updated to preserve analyses whenever possible, and frequently trigger recomputation of all analyses whenever they change the IR. Thus, the extra analysis invalidation machinery becomes less useful, and the increased overhead appears as the slowdown.

7.2.2 Package Image Compilation

As a slightly different workload, package image optimization times were also measured for both the legacy and new pass managers. Here, the measurements were captured by running import Pkg; Pkg.add("OrdinaryDiffEq"); with a nonexistent ~/.julia folder and a single precompilation task. The package images were all compiled at optimization level O3. The results from the packages with median optimization times being greater than 1 second are depicted in figures 7-3 and 7-4.

Figure 7-3 shows that all of the 8 packages with median optimization times greater than 1 second observed speedups with the new pass manager compared to the legacy pass manager. Figure 7-4 more quantitatively shows that LoopVectorization and LinearSolve had the smallest speedups between 2.5% to 3%, while ExponentialUtilities and NonlinearSolve had the largest speedups near 15% to 17.5%. Interestingly, both LoopVectorization and LinearSolve had fairly large variances in the timing reported, which could indicate that their true speedups are higher than what was measured.

Overall, this indicates that integration of the new pass manager into image compilation would result in a net speedup of user code. Given that at O3 the JIT compiler also saw speedups from migrating to the new pass manager, and that most user code is run at O3, integration of the new pass manager into the JIT compiler should also result in some reduction in compilation time.

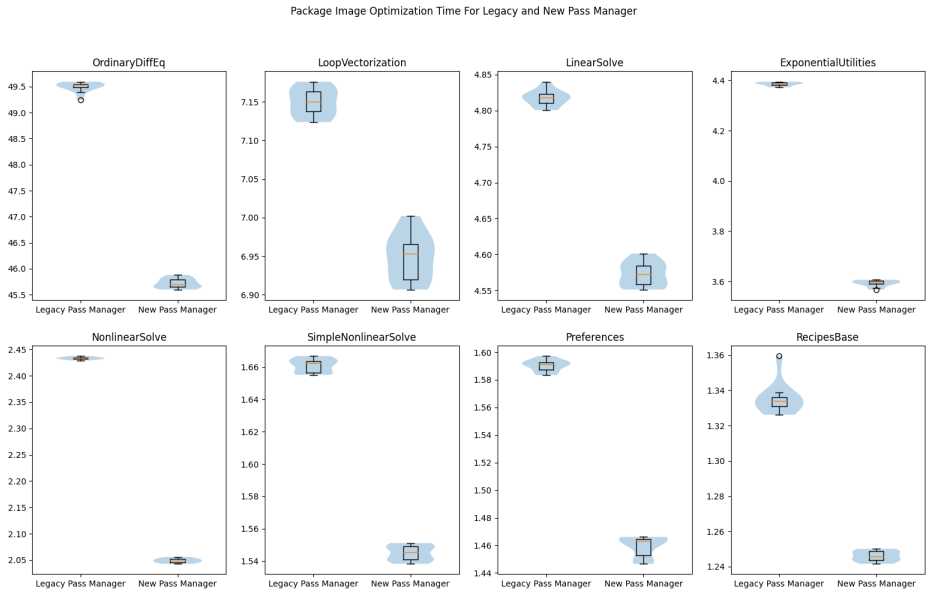

Figure 7-3: Pass Manager Timing Comparison for Package Image Construction

Package images display a consistent reduction in time spent optimizing when using the new pass manager as compared to the legacy pass manager. Unlike figure 7-1, neither baseline nor scaling is consistent among subplots, due to small differences in large total optimization times.

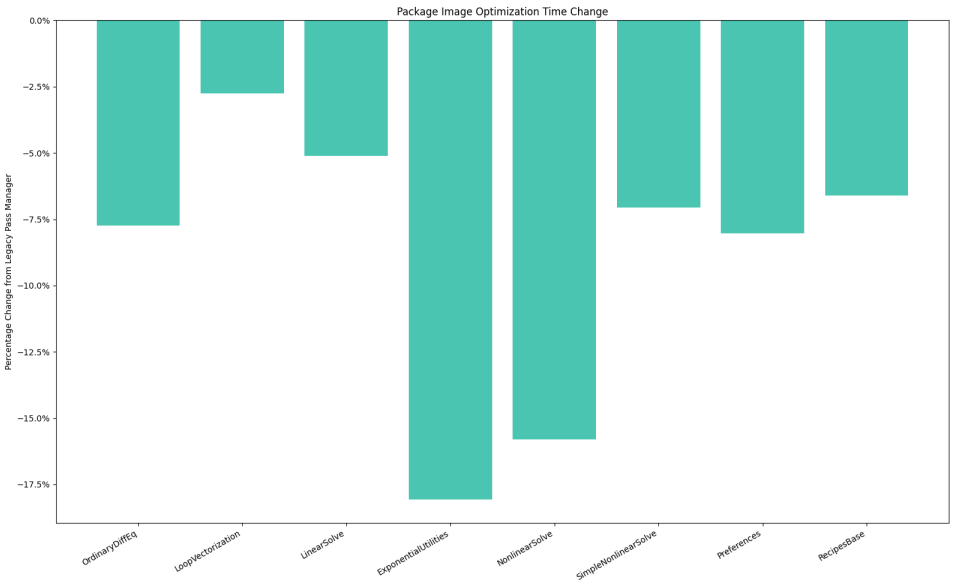

Figure 7-4: Pass Manager Performance Comparison for Package Image Construction

Package images show a consistent improvement in optimization time, ranging from 2.5% to 17.5%. Packages are ordered left to right by decreasing median optimization time.

7.3 Further Optimizations

7.3.1 Improved LLVM Analysis Preservation

Currently, as mentioned in section 7.2.1, many of the LLVM passes written for Julia specifically do not attempt to preserve LLVM analyses. When other LLVM passes need to use those analyses, they must be recomputed from scratch, at much higher cost compared to keeping the existing analysis up to date. By updating the Julia-specific LLVM passes to preserve additional analyses, it may be possible to improve the relative performance of the new pass manager at O0. In addition, these improvements would also help reduce the time spent at O3, where the likelihood of a downstream pass requiring an analysis is higher.

7.3.2 Pass Pipeline Nesting Optimization

One of New PM's benefits is that it makes explicit the nesting between module, call graph, function, and loop passes. Crossing this nesting boundary has implications on when passes are run on particular IR units, and it is typically preferred to group related passes close together to maximize cache speedups. Currently Julia's pass pipeline has not been optimized for this grouping, which will likely provide large speedups when compared to the old pass manager where the nesting was implicit.

7.3.3 Pass Pipeline Updating

With each new version, LLVM adjusts its default pass pipelines for different optimization levels to have different combinations of passes and different options on those passes. However, LLVM's default pipelines are tuned for C and C++, rather than a more dynamic language such as Julia. Therefore, Julia maintains its own pass pipeline which must be kept up to date with LLVM versions. After the transition to New PM, it would make sense to reevaluate the pass pipeline to account for LLVM's changes and identify any low-hanging fruit for either generated code improvements or compile time improvements from replacing complex passes with simple passes.

Chapter 8

Discussion and Conclusions

8.1 Summary of Results

Of the different methods used to improve the Julia compiler's performance, the largest speedups come from parallelization. Intuitively this makes sense, as parallelization of compiler work opens up opportunities to perform vastly more work in the same span of real time. However, the results from chapter 6 and chapter 7 show that there are still opportunities to reduce the work performed by the compiler during compilation.

These optimizations also build upon each other in different ways. Lazier compilation has a multithreading impact because functions are optimized and emitted as machine code when the function is first called. Without laziness, this optimization and machine code emission happens inside the global compilation lock; with laziness, they happen outside the lock. Combining lazier compilation with the concurrent type inference discussed in chapter 5 would likely increase the speedup for concurrently compiling functions.

Likewise, the new pass manager reduces the amount of time spent in the optimization phase, benefiting benchmarks in both concurrent type inference and parallel image compilation. However, this speedup would be mitigated by the fact that optimization tends to be in the parallel fraction of work rather than the serial fraction, but the work would allow a more enhanced pass pipeline to further improve the performance of the generated code, without compromising compiler performance.

8.2 Sources of Error and Limitations

All of the experiments conducted here were performed with 10 trials at each combination of parameters. However, some of the experiments, especially with regards to serial type inference on multiple threads, still exhibited significant variability between trials. For concurrent type inference in particular, the use of spinlocks hampers repeatable testing, as high CPU usage can trigger a number of deoptimizations such as thermal throttling or lower dynamic voltage frequency scaling. In other cases, the machine's active processes were monitored by use of the command top, but the top command itself could add a small amount of noise to the measurements.

All measurements were performed on a x86-64 machine with two 32-core AMD EPYC 7502 CPUs running Ubuntu 18.04 or Ubuntu 22.04. The full output of lscpu can be found in appendix TODO.

8.3 Applicability of Performance Enhancements

8.3.1 Current Merge Status

Much of the groundwork for the changes have been merged into Julia 1.9 or the upcoming Julia 1.10 release. In particular, parallel system image generation will be shipped with Julia 1.10. However, there are some remaining blockers to full implementation of each of these features.

Julia's package manager already has parallel capabilities, but these parallel capabilities are limited to precompiling packages that do not depend on each other in parallel. The proposed work enables parallelism within precompilation of packages, but balancing parallelism between multiple packages and within each packages remains unimplemented. However, preliminary results suggest that a noticeable speedup is still achieved by including intra-package parallelism even when inter-package parallelism is enabled.

Concurrent type inference is mostly enabled, but it is difficult to trigger its appearance in Julia 1.9 due to a particular acquisition of the global compilation lock.

While this statement was altered for testing purposes, in order to commit to moving that acquisition so as to permit concurrent type inference additional verification of thread safety in the surrounding code must be achieved.

LLVM's new pass manager has a few remaining performance issues in generated code when compared to the legacy pass manager, mostly around loop unrolling and loop unswitching, which were heavily revamped in the new pass manager. Once those issues are resolved, Julia will switch over to using the new pass manager, as the legacy pass manager implementation is slated to be deprecated and unmaintained by LLVM upstream.

Lazier compilation remains the feature that has the longest timeline for inclusion, as it depends on using the JITLink linker, which currently does not have full feature or platform parity with the current RtDyld linker. Furthermore, the implementation of lazier compilation described here may not end up being the implementation that is merged, as external improvements to the compiler could enable other parts of the compilation process to be included in lazier compilation, such as codegen itself. In that situation, the benchmarks for improved laziness would likely improve, as less upfront work would be necessary, and additional work would be happening outside the global compilation lock.

8.3.2 Future Impact

As chip designers are actively adding more parallel processing capabilities to their products, the impact of the parallelization work will compound in the future as consumer devices obtain more cores. As the parallelization work also has the largest speedups, compared to the lazier compilation and new pass manager, this increase in parallel capabilities will become invaluable to users in the future.

On the serial optimizations, a tiered JIT compiler would enable optimizing for both compile time and runtime on the appropriate methods. Bringing a tiered JIT compiler to Julia would likely bring widespread reductions in compile time, to a greater extent than just lazier compilation alone. Lazier compilation's main benefit in this regard is to set up the groundwork for building a tiered JIT compiler.

The new pass manager's benefit is likely to be a one-time speed boost, though its infrastructure improvements may enable small optimizations in the future. Out of all of the improvements discussed here, the new pass manager is therefore likely to have the smallest future impact.

Bibliography

[1] Java Virtual Machine Guide.

[2] Proper maintenance and care of multi-threading locks · The Julia Language, December 2022.

[3] System Image Building · The Julia Language, March 2023.

[4] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. Association for Computing Machinery. event-place: Atlantic City, New Jersey.

[5] Julia Belyakova, Benjamin Chung, Jack Gelinas, Jameson Nash, Ross Tate, and Jan Vitek. World age in Julia: optimizing method dispatch in the presence of eval. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–26, November 2020.

[6] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B. Shah, Jan Vitek, and Lionel Zoubritzky. Julia: dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–23, October 2018.

[7] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[8] Daniele Cono D'Elia and Camil Demetrescu. On-stack replacement, distilled. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 166–180, Philadelphia PA USA, June 2018. ACM.

[9] Arthur Eubanks. The New Pass Manager, March 2021. Section: posts.

[10] John L. Gustafson. Reevaluating Amdahl's Law. *Commun. ACM*, 31(5):532–533, May 1988. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[11] Tim Holy. Tutorial on the foundations · SnoopCompile, January 2023.

[12] Tim Holy, Jeff Bezanson, and Jameson Nash. Analyzing sources of compiler latency in Julia: method invalidations, August 2020.

[13] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004.

[14] Romolo Marotta, Davide Tiriticco, Pierangelo Di Sanzo, Alessandro Pellegrini, Bruno Ciciani, and Francesco Quaglia. Mutable Locks: Combining the Best of Spin and Sleep Locks, June 2019. arXiv:1906.00490 [cs].

[15] Colin Taylor, Satyanarayana G. Manyam, and David Casbeer. On-board implementation using Julia precompiler for rendezvous path planning. In *AIAA SCITECH 2023 Forum*, National Harbor, MD & Online, January 2023. American Institute of Aeronautics and Astronautics.

[16] Roland Westrelin. How the JIT compiler boosts Java performance in OpenJDK, June 2021. Section: Kubernetes.