Speeding up the Local C++ Development Cycle with Header Substitution

Nader Al Awar
University of Texas at Austin
Austin, USA
nader.alawar@utexas.edu

Zijian Yi
University of Texas at Austin
Austin, USA
zijianyi@utexas.edu

George Biros
University of Texas at Austin
Austin, USA
biros@oden.utexas.edu

Milos Gligoric
University of Texas at Austin
Austin, USA
gligoric@utexas.edu

Abstract

C++ remains one of the most widely used languages in various computing fields, from embedded programming to high-performance computing. While new features are constantly being added to C++, an important aspect of the language that is often overlooked is its compilation time. Merely including a few header files can cause compilation time to increase significantly. An alternative to including header files is using forward declarations; however, the rules for forward declaring classes and functions are non obvious and confusing to most developers. Additionally, forward declaring methods, as well as functions that accept lambdas as arguments, is not possible. In this paper, we present a novel technique, termed Header Substitution, to automatically detect opportunities for forward declarations with the goal of replacing includes of header files and improving compilation time. Header Substitution also introduces function wrappers as an alternative to forward declaring methods and functions with lambda arguments. We implemented Header Substitution in a tool, dubbed Yalla, and applied it to various C++ projects in order to speed up the development cycle, i.e., the debugging, editing, compiling, and rerunning loop, achieving up to a 24.5× speedup when compiling C++ files and a 4.68× speedup of the development cycle.

CCS Concepts: • **Software and its engineering** → **Incremental compilers**; **Source code generation**.

Keywords: C++, compilation, Header Substitution

ACM Reference Format:
Nader Al Awar, Zijian Yi, George Biros, and Milos Gligoric. 2025. Speeding up the Local C++ Development Cycle with Header Substitution. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3696443.3708942

1 Introduction

The local development cycle is a universal concept in software development. Prior to deploying software to production, developers clone the code to their local machine and iterate over it by editing, compiling, and debugging it, as shown in Figure 1. This cycle is ubiquitous across many software development tasks, including bug fixing, prototyping, and performance tuning [18, 20, 30].

The main benefit of developing locally is getting instantaneous feedback on changes made to the code. Therefore, in order to increase developers' productivity, it is necessary to ensure that the local development cycle proceeds rapidly. For compiled languages, the bottleneck in this cycle is frequently the compilation step. C++ in particular is known to suffer from long compile times which are disruptive to the developer during the local development cycle [25, 29, 34, 35].

As C++ was designed to be backwards compatible with the C language [31], C++ adopted the C include system, where developers divide their code into source and header files. Code that is meant to be shared across multiple files is placed in header files, which can then be included in other source or header files. In this system, re-compiling a source file requires re-compiling all included header files. Since these header files could include other header files transitively, compilation times can grow sharply during the development cycle as more files are included. This can be frustrating to programmers as they must pay the cost of compiling all included headers for even minor modifications made to unrelated parts of their code.

C++ programmers must therefore ensure that their headers remain relatively lightweight. This can be achieved by including in the headers only the code necessary to interface

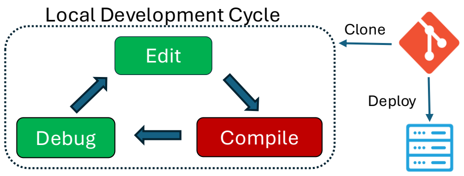


Figure 1. The universal local development cycle. In compiled languages, compilation time is downtime for developers, so we aim to speed it up in this paper.

with other parts of the program, while moving the rest into source files. However, this is not always possible, as programmers might be using headers from third party libraries written by others. These can be especially problematic in the case of header-only libraries [12] which include an entire library's implementation in the user's source code.

Instead of including a header file, an alternative that can improve compilation time is to use *forward declarations* [13]. Programmers can avoid including header files by adding function and class declarations defined in those header files directly to their source files. However, this approach has some drawbacks: first, the rules for forward declaration are not straightforward and are hard to get right, which could result in hard to debug compilation and linking errors. Second, forward declarations do not work well with C++ templates and do not work at all with lambdas passed as template arguments and class methods, all of which are extensively used in modern C++ code.

Another technique to improve compilation time is to use pre-compiled header files [11], or PCH for short. This enables the compiler to parse headers beforehand so that compiling any file that includes them later on is faster. However, this only saves the cost of parsing the header files and not the later stages of compilation, such as optimization and machine code generation. Another potential solution is to use modules, a recent C++ feature introduced in C++20 [9]. Support for modules is still lacking in major C++ compilers, and it remains to be seen whether they will be widely adopted by the C++ community [5].

We present *Header Substitution*, a technique for automatically replacing included header files in C++ source code with forward declarations, applied locally on a developer's machine to speed up the development cycle. Header Substitution replaces include statements in source files while guaranteeing that the code still compiles and runs correctly. It makes extensive use of forward declarations for classes and functions, even templated ones. It also relies on code generation to allow forward declaration for templates, templates with lambdas, and class methods. Header Substitution examines the user's source code to determine which parts of the included header files they actually use, generating a lightweight header file much smaller in size than the header being substituted.

We implemented Header Substitution in a tool named YALLA and evaluated it on existing C++ projects, including PyKokkos [16, 17], a Python framework for scientific computing that generates and compiles C++ Kokkos code [32], as well as a number of widely used C++ libraries, including RapidJSON [3], OpenCV [2], and Boost.Asio [1].

The main contributions of this paper include:

⋆ Design of Header Substitution, a technique for replacing header files that are costly to compile in C++ source files.

⋆ Implementation of header substitution in a tool, dubbed YALLA, for header substitution in C++.

⋆ Evaluation of YALLA on C++ generated by PyKokkos, as well as other popular C++ libraries. Our results show that YALLA improves compilation time of C++ code by 24.5× on average, significantly speeding the development cycle, even when compared to using PCH files.

The source code for YALLA is available at https://github.com/EngineeringSoftware/llvm-project-yalla/tree/yalla/clang-tools-extra/clang-yalla.

2 Background

In this section, we illustrate the C++ compilation process (Section 2.1) and introduce techniques used to speed it up (Section 2.2). We then motivate the need for faster C++ compilation in the development cycle (Section 2.3).

2.1 C++ Compilation

We illustrate the stages followed by most C++ compilers using the example in Figure 2, focusing mainly on the preprocessor phase and template instantiation. Figures 2a and 2b show a C++ source file that includes a header before and after the template instantiation phase respectively. Figures 2c and 2d show the same example with forward declarations, which we discuss in the next section in detail.

The first step is passing the C++ source file (shown in Figure 2a) to a compiler, which reads the file into memory and removes comments. It then moves to the *preprocessor* phase, which handles lines that begin with the # character, such as `#include`, which can be seen on line 2 in Figure 2a. The preprocessor searches for a header file (typically ending with a `.h` or `.hpp` extension) with that name in a set of default and user-defined search paths. Once found, the preprocessor replaces the `#include` with the contents of the found header file. This process is repeated recursively on the included header files until all directives have been processed. The output of this stage is known as a *translation unit.*

The next phases include lexical, syntactic, and semantic analyses, followed by *template instantiation.* Templates are a C++ feature that enable generic programming, i.e., writing definitions in terms of types that will be specified later. Line 9 in Figure 2a is an example of a function template. Other types of templates in C++ are class and variable templates (not

```
1  // main.cpp
2  #include "add.hpp"
3
4  int main() {
5    g_add<int>(1, 2);
6  }
7
8  // add.hpp
9  template<typename T>
10 T g_add(T x, T y) {
11   return x + y;
12 }
```

(a) The sample source and header files.

```
1  template <typename T>
2  T g_add(T x, T y) {
3    return x + y;
4  }
5  template<>
6  int g_add<int>(
7   int x, int y) {
8    return x + y;
9  }
10 int main() {
11   g_add<int>(1, 2);
12 }
```

(b) The source file after template instantiation.

```
1  // Forward declaration
2  template<typename T>
3  T g_add(T x, T y);
4
5  int main() {
6    // Template usage
7    g_add<int>(1, 2);
8  }
```

(c) Forward declaration of a templated function.

```
1  // Template definition
2  template<typename T>
3  T g_add(T x, T y) {
4    return x + y;
5  }
6
7  // Explicit instantiation
8  template
9  int g_add<int>(
10  int x, int y);
```

(d) Template definition and explicit instantiation.

Figure 2. An example illustrating the C++ compilation process and forward declarations.

shown here). A generic template definition by itself does not cause any code to be generated [10]. Rather, the template must be *instantiated* by setting the generic types to concrete ones, typically when the template is used (line 5 in Figure 2a). The compiler generates a different version of the template for every combination of concrete types used. The output of this stage can be seen in Figure 2b.

The final phases typically include optimization, object code generation, assembly, and linking. During linking, the linker searches for missing definitions of symbols such as classes and functions in other object code files.

2.2 Forward Declarations

The C++ include system can result in a relatively slow compilation process. Forward declaration means declaring classes or functions originally defined in other files. This can be used instead of including those files to speed up compilation.

Libraries typically provide a few header files containing most of their public API, which is problematic for several reasons. First, these headers can contain definitions instead of just declarations, meaning that they are larger than they need to be and costly to compile. Separating definitions into different source files requires more effort on the library developer's part and can even be impossible for templates. Second, the library header itself typically includes more files, including standard library and system headers, so simply including one file results in many more files being included transitively, further bloating the library header. Third, even if the user's code needs a small part of the API, it must include the entire library header, so developers end up paying the cost of compiling the whole file.

An alternative to including a header file is to *forward declare* classes and functions needed from that header. A forward declaration of a class or function makes it visible to the user's code without including the header file it was originally declared in. Forward declaring a function means specifying its return type, name, and parameters but not the function body. Lines 2 and 3 in Figure 2c show an example of a forward declared function as a substitute for including the header file add.hpp. Its definition can be seen on lines 2 to 5 in Figure 2d. Forward declaring a class means simply specifying its name while not providing any of its fields or methods. Missing class and function definitions will be linked to the user's code at the end during the linking stage.

While the example shown in Figure 2 shows a case where a forward declaration can easily replace a header include, the two are not always functionally equivalent. Forward declared classes are *incomplete types* and are not semantically equivalent to class definitions. Since a forward declared class does not list its fields and methods, any later references to them results in a compilation error. Additionally, since the compiler does not know the memory layout of the class, it can only be used as a reference or pointer (except in function declarations). The rules for forward declared classes and incomplete types are therefore non-trivial and require significant effort from the developer to implement.

Further complications exist for templates. As with non-templated code, adding a forward declaration is necessary (line 2 in Figure 2c), but not sufficient. The compiler will only generate code from a template definition once it sees a usage of the template. Separating the template definition (line 2 in Figure 2d) and the template usage (line 7 in Figure 2c) into different source files means that the compiler will not generate the code in either file. It will not generate it in the definition file since it is not aware of the usage in the other file. Additionally, it cannot generate it in the usage file since it does not know what the definition is. The solution is to *explicitly instantiate* the function in the file containing the definition using the desired template arguments (line 8 in Figure 2d), forcing the compiler to generate code with those template arguments. However, this is not feasible for library developers, as it requires them to know all possible combinations of template arguments that users may need for each template beforehand.

One other effect of separating definitions from usages is that it impedes optimization since compilers optimize each translation unit separately. *Link time optimization (LTO)* optimizes code across translation units during the linking phase, partially alleviating this issue.

2.3 PyKokkos

PyKokkos [17] is a Python framework for writing performance portable, high-performance kernels, i.e., code that runs on different hardware architectures, such as CPUs and GPUs, efficiently. When the user calls a kernel, PyKokkos translates the user-written Python kernel code into C++ and Kokkos [24] and generates language bindings to enable interoperability between the two languages. Since the generated kernel uses the C++ Kokkos library, PyKokkos adds an `#include <Kokkos_Core.hpp>` statement in the containing file. PyKokkos then invokes a compiler to compile the generated C++ code into a module that it imports from Python.

When the user makes code changes to the kernel, PyKokkos recompiles the entire generated code, including large parts of Kokkos due to the `#include <Kokkos_Core.hpp>` statement. The user thus incurs the same compilation cost as the initial compilation every time the kernel is changed, severely slowing down the PyKokkos development cycle.

3 Header Substitution

In this section, we illustrate Header Substitution by applying it to a C++ source file generated by PyKokkos. First, we show the C++ source file, and briefly explain the input and intended output which will lead to faster compilation times (Section 3.1). We then show the steps needed for Header Substitution, first through forward declarations (Section 3.2) and then through code transformations (Section 3.3). We then introduce function and method wrappers and show how they are defined and instantiated (Section 3.4). Finally, we present the full algorithm for Header Substitution (Section 3.5).

3.1 PyKokkos Kernel

Figure 3 shows an example of the C++ Kokkos code generated by PyKokkos, split across two files, `functor.hpp` and `kernel.cpp`. The former contains the kernel's signature while the latter contains its definition. In order to use Kokkos, the code includes the main Kokkos header (line 3). It also has a `struct` (line 12) containing the arguments used by the kernels as member variables. The first variable (line 14) is a simple integer scalar and the second (line 15) is a Kokkos `View`, the Kokkos multidimensional array type. Views include their datatype, dimensionality, and array layout (row or column major) as template arguments. This view is a two-dimensional integer array (`int**`) that uses the `LayoutRight` memory layout. The rest of the code contains the kernel signature (line 17) in `functor.hpp` and the kernel definition (line 22) in `kernel.cpp`. Note that in Kokkos, the kernel is defined as the overloaded call operator `operator()`.

```
1  // ** functor.hpp **
2  // Library header include
3  #include <Kokkos_Core.hpp>
4
5  // Type aliases
6  using sp_t = Kokkos::OpenMP;
7  using member_t =
8  Kokkos::TeamPolicy<sp_t>::member_type;
9  using Kokkos::LayoutRight;
10
11 // Functor definition
12 struct add_y {
13  // Member variables (fields)
14  int y;
15  Kokkos::View<int**, LayoutRight> x;
16  // Member functions (methods)
17  void operator()(member_t &m);
18 };
19
20 // ** kernel.cpp **
21 // Member function definition
22 void add_y::operator()(member_t &m) {
23  int j = m.league_rank();
24  Kokkos::parallel_for(
25   Kokkos::TeamThreadRange(m, 5),
26   [&](int i) { x(j, i) += y; });
27 }
```

Figure 3. C++ Kokkos code that includes `Kokkos_Core.hpp`, an expensive header. `functor.hpp` includes a class which declares the kernel by overloading the call operator while `kernel.cpp` contains the kernel's definition.

As mentioned previously (Section 2.2), the Kokkos header file contains the entire Kokkos API, of which only very little is used here. Thus, the goal of Header Substitution is to avoid including the entire file. However, simply removing the `#include` statement is not enough as it results in compilation errors due to the Kokkos symbols no longer being visible, so we apply Header Substitution to Figure 3, resulting in Figure 4, which we will explain in this section. Note that developers do not need to worry about the code in Figure 4 as it is generated and maintained automatically.

3.2 Forward Declarations

At this stage, the compiler only needs the symbols to be visible in the code, so it is enough to only move the declarations, i.e., *forward declare* the symbols (Section 2.2) to make them visible at compile time. Certain functions require the introduction of *function wrappers*, which provide an extra layer of indirection needed to call those functions. Together, function forward declarations and wrappers form a lightweight header that serves as a substitute for the original header. Figure 4a shows the lightweight header that replaces the original header.

```
1  // ** lightweight_header.hpp **
2  namespace Kokkos {
3   // Forward declared classes
4   class OpenMP;
5   template<...> class View;
6   class LayoutRight;
7   template<...> class HostThreadTeamMember;
8  }
9
10 // Function wrappers
11 template<...>
12 TeamThreadRangeBoundariesStruct*
13  TeamThreadRange_w(...);
14 template<...>
15 void parallel_for_w(
16 TeamThreadRangeBoundariesStruct*,..);
17 template<class ObjectT>
18 int league_rank(ObjectT&);
19 template<class ObjectT>
20 int& paren_operator(ObjectT&, int, int);
21
22 // Functor replacing lambda
23 struct lambda_functor {
24  int j, y;
25  Kokkos::View<int**, LayoutRight>* x;
26  void operator()(int i) const {
27   paren_operator(x, j, i) += y;}
28 };
```

(a) New lightweight header file.

```
1  // ** functor.hpp **
2  // New header include
3  #include <lightweight_header.hpp>
4
5  // Type aliases
6  using sp_t = Kokkos::OpenMP;
7  using member_t =
8  Kokkos::HostThreadTeamMember<sp_t>;
9
10 struct add_y {
11  int y;
12  Kokkos::View<int**, Kokkos::LayoutRight> *x;
13  void operator()(member_t &m);
14 };
15
16 // ** kernel.cpp **
17 void add_y::operator()(member_t &m) {
18  int j = league_rank(m);
19  parallel_for_w(
20   *TeamThreadRange_w(m, 5),
21   lambda_functor{x, j, i});}
```

(b) Modified C++ Kokkos code.

Figure 4. The result of applying Header Substitution to the code in Figure 3. `lightweight_header.hpp` provides forward declarations of functions, classes, and function wrappers, while `functor.hpp` and `kernel.cpp` are modified to use the new forward declarations and wrappers.

3.2.1 Classes.

In Figure 3, the Kokkos classes that must be forward declared are `OpenMP`, `View`, `LayoutRight`, and `TeamPolicy<sp_t>::member_type`.

The actual classes to be forward declared are the ones after the last scope operator `::`, e.g., `OpenMP`. Symbols that appear before a scope operator are either classes or, in this case, namespaces (Kokkos). The forward declarations for `Kokkos::OpenMP`, `Kokkos::View`, and `Kokkos::LayoutRight` are straightforward and are shown on lines 4 to 6 in Figure 4a (template parameters are omitted due to space limitations). Forward declaring a symbol in a namespace requires defining the namespace (line 2) before declaring the symbol.

The forward declaration for `Kokkos::TeamPolicy<sp_t>::member_type` is more involved. Here, the symbol before the last scope operator is a class (`TeamPolicy<sp_t>`) and the class to be forward declared is `member_type`, referred to as a *nested class* since it is defined *inside* another class. As we rely on forward declaring classes extensively, and forward declaration means that the definition is omitted, the containing classes cannot be defined and the nested class cannot be forward declared. In other words, forward declaring nested classes is not possible and Header Substitution does not currently support them. In our example, Kokkos defines `member_type` as a type alias to `HostThreadTeamMember` that is not nested so we forward declare that instead.

3.2.2 Functions.

The other symbols to be forward declared are functions. In Figure 3, the Kokkos functions used are `TeamThreadRange()` and `parallel_for()`.

Forward declaring `Kokkos::TeamThreadRange()` directly results in a compilation error since its return type is `Kokkos::Impl::TeamThreadRangeBoundariesStruct`, a class defined in the Kokkos header file. The intuitive solution is to forward declare this class as well, making it visible to the compiler albeit as an incomplete type. However, compilation still fails since the return type is now incomplete and the compiler does not know its memory layout. In other words, forward declaring functions with incomplete return types returned by value (i.e., not pointers or a references) is not possible.

To account for functions with incomplete return types, we introduce *function wrappers*. Function wrappers are an additional layer of indirection that allow us to call functions that would otherwise be inaccessible due to incomplete types. Given some function f, its function wrapper g has the same signature except for the return type, which will be a pointer to the original return type. Internally, g dynamically allocates an object on the heap using the `new` operator, passing the value returned by the original function f as an argument. Dynamic allocation on the heap is necessary because the object would be destroyed once the function resolves if it were allocated on the stack. Lines 11 to 13 in Figure 4a show the function wrapper declaration.

Similarly, we require function wrappers for functions that take incomplete types by value as arguments, converting those types to pointers in the wrapper's signature. One such function is `Kokkos::parallel_for()` which requires the value returned by `TeamThreadRange()` to be passed by value as the first argument. Since we introduced a wrapper that returns a pointer instead, we add a wrapper for `parallel_for` that has the corresponding parameter be a pointer type as shown on line 16 in Figure 4a.

3.2.3 Methods. Class methods are also handled at this stage. There are two methods in Figure 3, `league_rank()` and and `x(j, i)` called on lines 23 and 26 respectively (note that the latter is an overloaded call operator, equivalent to `x.operator()(j, i)`). All methods must be declared within class definitions, meaning that the methods of forward declared class cannot be used. In other words, forward declaring methods of forward declared classes is not possible.

To solve this issue, we introduce *method wrappers*. Like function wrappers, method wrappers also add an extra layer of indirection. However, their purpose is to call methods that are otherwise inaccessible due to their classes being forward declared. Method wrappers are similar to function wrappers but add a parameter to the original signature to accept the class instance as an argument, which we then use to call the original method within the body of the wrapper (similar to the implicit `this` introduced to methods by C++ compilers).

The method wrappers for `league_rank()` and `x(j, i)` are shown on lines 18 and 20 respectively in Figure 4a. The first argument is parameterized on the type of the class containing the method. The rest of the arguments match the original method.

3.3 Code Transformations

After adding the forward declarations, function wrappers, and method wrappers, the next step is to modify the original code to integrate the new additions, as compiling the code at this stage would result in compilation errors due to the rules concerning forward declared classes and incomplete types, the function and method wrappers, and other issues.

3.3.1 Include. Since the goal of Header Substitution is to replace header includes, the first transformation is replacing the include statement of the expensive header with an include of the lightweight header (line 3 in Figure 4b).

3.3.2 Classes. Since all Kokkos related classes at this stage are forward declared, which makes them incomplete types, we must modify the source file to account for this. Recall that the compiler cannot know the memory layout of incomplete types (Section 2.2). Therefore, simply replacing class definitions with forward declarations is not enough: we must replace all direct usages of these classes with pointers or references. Since we are using dynamically allocated objects through our wrappers, we will use pointers.

An example of usage of a class that Header Substitution makes into an incomplete type is on line 15 in Figure 3. Header Substitution replaces this declaration with a pointer to the type instead of just the type itself, i.e., it adds `*` directly after the type. Line 12 in Figure 4b reflects this change.

3.3.3 Functions. Calls to forward declared functions do not need to be modified. However, for each function and method wrapper introduced by Header Substitution, we replace the original function calls with calls to the wrappers. At the call site, we change the function being called from its original name `Kokkos::TeamThreadRange` (line 25 in Figure 3) to the wrapper's name `TeamThreadRange_w` (line 20 in Figure 4b). We also do the same for the other function with a wrapper, `Kokkos::parallel_for()`.

3.3.4 Methods. Methods of forward declared classes cannot be referenced as they are not visible (Section 3.2.3). Instead, we introduced method wrappers.

In Figure 3, there are two method calls: `x(j, i)` and `team_member.league_rank()`. Two changes are needed: replacing the original name with the wrapper name and passing the class instance as the first argument. In this case, we replace `x(j, i)` with `paren_operator(x, j, i)` (line 27 in Figure 4a) and `team_member.league_rank()` with `league_rank(team_member)` (line 18 in Figure 4b).

3.4 Wrapper Definition and Instantiation

The only remaining task is to provide the definitions for all the new function and method wrappers introduced previously. This is necessary because so far we have only provided *declarations* for these new functions but not *definitions*.

The first question is *what* the definitions will contain. Since the goal of these wrappers is to provide a layer of indirection, the wrappers must call the original function within their body. This leads to another question: *where* will the wrappers be defined? Defining them in the same file as the kernel is not viable because the definitions will call Kokkos functions, which requires including the Kokkos header `Kokkos_Core.hpp`. The solution is to add the definitions to a new source file that includes the Kokkos header, then compile and link it with the kernel file.

Compiling the separate file with the definitions results in no code being generated. Recall that compilers only generate code from templates at the site where they are used (Section 2.2). If a file only contains templated function definitions and not usages of or calls to these functions, then the compiler will not instantiate the wrappers when compiling the wrappers. The solution is to use *explicit template instantiation*, a technique which forces the compiler to instantiate the wrapper with the specified template arguments. We therefore explicitly instantiate each wrapper with the types used at call sites in Figure 4b.

Explicit instantiation of `parallel_for_w` is more complex. Looking at the call site on line 24 in Figure 3, we can see that the third argument is a lambda expression, an anonymous function object with a unique type that is specified

Table 1. Summary of header substitution rules that show how each C++ symbol is transformed.

Symbol Types	Code Transformations
Class or struct	Forward declare and replace usages with pointers.
Type alias	Resolve and forward declare.
Enum	Replace usages with the datatype of the size of the enum.
Function	Forward declare if it does not use forward declared classes. Otherwise create a wrapper and replace usages with calls to the wrapper.
Class method & field	Create wrapper with class type as the first argument. Replace usages with call to wrapper, passing the object as the first argument.
Lambda	Create an equivalent functor that overloads the call operator and then replace the usage with a call to the functor's constructor.

by the file and line in which it is defined, meaning it is not possible to explicitly write out the type of the lambda. Therefore, it is not possible to explicitly instantiate a templated function using a lambda as a template argument.

C++ lambdas are functionally equivalent to function objects, also known as functors [8]. A functor is a C++ class that overloads the call operator (`operator()`). We have already seen an example of a functor in Figure 3 on line 12. We will therefore replace the lambda by generating a new functor and overloading the call operator. The new functor is shown in Figure 4b on line 23, and the lambda expression is replaced with a call to the functor constructor shown on line 21. All that is left is to explicitly instantiate `parallel_for` using the newly generated functor as a template argument.

3.5 Header Substitution Algorithm

A summary of Header Substitution rules is in Table 1.

We will now present a formalized version of Header Substitution in Figure 5. The input is `sources`, the paths to the C++ files from which we must replace an include statement, and `header`, the path to the included C++ header file. The example shown above illustrates Header Substitution on Kokkos code but this technique works for any C++ code.

The first phase of the algorithm is the analysis phase, during which the `sources` and `header` are parsed, and the used functions, classes, and lambdas are extracted (lines 2 to 10). This step involves getting the used symbols (classes and functions) in `sources` and the defined symbols in `header`, and finding the common symbols between them to determine which symbols from the `header` file are used in `sources` (lines 3 and 5). For classes, we also check whether they are type aliases to other classes and use those instead (line 4). Additionally, all classes referenced in the used functions' signatures are also added to the list of used symbols, as they appear in the functions' forward declarations (lines 7 to 10).

```
Require: sources- The input source files
Require: header- The header to be substituted
 1: function SUBSTITUTEHEADER(sources, header)
 2:     analyzer ← Analyzer(sources, header)
 3:     usedClasses ← analyzer.getUsedClasses()
 4:     analyzer.resolveAliases(usedClasses)
 5:     usedFunctions ← analyzer.getUsedFunctions()
 6:     lambdas ← analyzer.getLambdas()
 7:     for function in usedFunctions do
 8:         newClasses ← getClassesInFunction(function)
 9:         usedClasses.append(newClasses)
10:     end for
11:     for class in usedClasses do
12:         makeClassForwardDeclarable(class, sources)
13:         forwardDeclareClass(class, sources)
14:     end for
15:     wrappers ← []
16:     for function in usedFunctions do
17:         if needsWrapper(function) then
18:             function ← createWrapper(function)
19:             wrappers.append(function)
20:         end if
21:         forwardDeclareFunction(function, sources)
22:     end for
23:     for lambda in lambdas do
24:         transformLambda(lambda, sources)
25:     end for
26:     replaceInclude(sources, header)
27:     writeWrapperFile(wrappers)
28: end function
```

Figure 5. The Header Substitution algorithm.

The second phase includes forward declaration and code transformation. This involves forward declaring classes (lines 11 to 14) and functions (lines 16 to 22). For functions requiring wrappers, a function wrapper is generated and forward declared instead (lines 17 to 20). Next, lambdas are transformed into functors (lines 23 to 25) to address the lambda template instantiation issue and the include statement is replaced (line 26). Finally, the wrapper definitions are written and instantiated in a separate file (line 27).

4 Implementation

In this section, we describe how we implemented Header Substitution in a tool called YALLA (Section 4.1), as well as its integration with the local development cycle (Section 4.2).

4.1 Yalla

We implemented Header Substitution in a standalone tool we named YALLA. YALLA provides an interface similar to the one shown in the algorithm in Figure 5: the user provides a source file and the header file they want substituted and

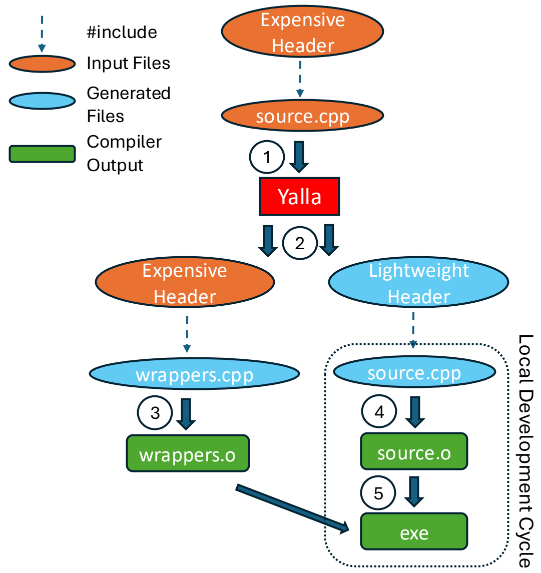


Figure 6. Yalla workflow and integration with the local development cycle.

Yalla does all the analysis and code generation. Yalla uses Clang [28], a compiler frontend for C/C++, to parse the code and build an abstract syntax tree (AST). It then uses Clang's AST Matcher library [15] to match the nodes representing the symbols in Table 1, while also using Clang's refactoring capabilities to implement the required changes.

Yalla goes over all nodes in the C++ AST to find definitions and usages. It looks for classes and functions defined in the header and used in the source file, as those will be forward declared later. Additionally, Yalla records the usage's *nature*, i.e., if the type is a pointer, reference, or a direct usage of the class. Yalla also records usages of lambdas.

After the analysis phase, Yalla generates the lightweight header and wrapper definitions, while also updating the input sources according to the rules in Table 1.

4.2 Integration with Development Cycle

We now describe how Yalla integrates with the local development cycle. Figure 6 shows the typical workflow of using Yalla. We assume a user has source files that include expensive headers they wish to substitute.

The user first passes the source files to Yalla by specifying the exact options needed to compile the code (step ① in Figure 6). Yalla applies Header Substitution by generating the modified source files (saved to different paths), the lightweight header that is included by the modified source files, and a wrappers file to hold the wrapper definitions (step ②).

The user then compiles the wrappers file into an object file `wrappers.o` (step ③). Since the wrapper definitions call the original functions, this file includes the expensive headers being substituted. This is as costly as compiling the original input files but in the context of the local development cycle it is only done once at the start since we assume that the wrappers and expensive header (i.e., the library) are rarely modified. Furthermore, we assume that the set of classes and functions used from the expensive header does not change, as this would require rerunning Yalla.

The user then compiles the modified source files into other object files `source.o` (step ④). As this only includes the lightweight header and the user's source code, this is much faster to compile than the user's original code. The final step is to link the object files to generate an executable (step ⑤).

After running Yalla and going through steps ① through ⑤ initially, the only steps needed to recompile the code during the development cycle are ④ and ⑤. The total number of lines of code being compiled is much smaller than what it was originally since Yalla replaces the expensive header with a lightweight one.

5 Evaluation

In this section, we present the results of our evaluation of Yalla. First, we show our experimental setup (Section 5.1) and the test subjects we use (Section 5.2). We then look at how well Yalla is able to improve compilation times compared to the default configuration and to using pre-compiled headers (PCH), while also looking at detailed compiler timers in Clang to understand why Yalla and PCH are able to improve compilation times (Section 5.3). We then examine how the improved compilation times speed up the development cycle (Section 5.4). Finally, we report the startup time of Yalla, i.e., the cost to run Yalla initially (Section 5.5).

5.1 Experiment Setup

We ran all experiments on an Ubuntu machine with an 8-core Intel i7-11700K 3.60GHz CPU and 64GB RAM. For all compilers, frameworks, and test subjects, we used the latest versions available at the start of the project. This includes Clang 15.0.6 to compile our test subjects, PyKokkos commit 3d4afd2, RapidJSON 1.1.0, OpenCV 4.10.0, and Boost.Asio 1.85.0. We obtain similar results with GCC 9.4.0 but due to space constraints we only show summarized results. The compiler flags we used are typically used by developers during the development cycle: we enabled all optimizations, all warnings, and debug info, e.g., `-Wall -Wextra -march=native -mtune=native -g -O3`. We measured wall clock compilation times using the Linux `time` command. All results reported are the arithmetic average of three runs.

5.2 Subjects

We used examples available from the repositories of the previously mentioned libraries as our test subjects. From PyKokkos, we used the 02, team_policy, nstream, and ExaMiniMD examples. These examples all generate kernels as C++ Kokkos code, with ExaMiniMD being a larger application that contains multiple such kernels. We use the

Table 2. Compilation time with Clang and speedup using Yalla and PCH.

File	Subject	Default [ms]	PCH [ms]	Yalla [ms]	PCH Speedup	Yalla Speedup
02	02	650	187	17	3.4×	**38.2×**
team_policy	team_policy	698	251	19	2.7×	**36.7×**
nstream	nstream	638	174	16	3.6×	**39.8×**
BinningKKSort	ExaMiniMD	663	194	17	3.4×	**39.0×**
FinalIntegrateFunctor	ExaMiniMD	639	189	17	3.3×	**37.5×**
ForceLJNeigh_for	ExaMiniMD	657	212	22	3.0×	**29.8×**
ForceLJNeigh_reduce	ExaMiniMD	651	200	22	3.2×	**29.5×**
InitialIntegrateFunctor	ExaMiniMD	656	196	18	3.3×	**36.4×**
init_system_get_n	ExaMiniMD	667	209	33	3.1×	**20.2×**
KinE	ExaMiniMD	651	191	17	3.4×	**38.2×**
Temperature	ExaMiniMD	649	194	16	3.3×	**40.5×**
archiver	RapidJSON	624	492	188	1.2×	**3.3×**
capitalize	RapidJSON	532	429	70	1.2×	**7.6×**
condense	RapidJSON	494	383	20	1.2×	**24.7×**
3calibration	OpenCV	1209	670	622	1.8×	**1.9×**
drawing	OpenCV	719	210	127	3.4×	**5.6×**
laplace	OpenCV	656	162	186	**4.0×**	3.5×
chat_server	Boost.Asio	2637	1835	277	1.4×	**9.5×**

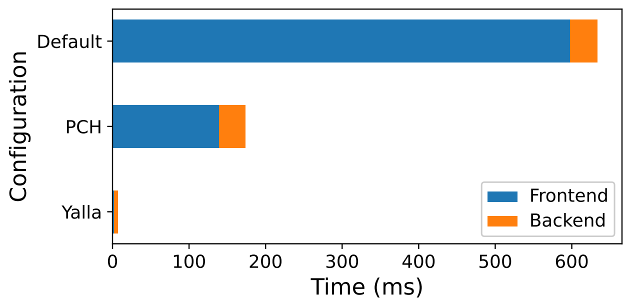
Table 3. Code statistics before and after applying Yalla showing total lines of code (LOC) and header files included.

File	Default LOCs	Yalla LOCs	Default Headers	Yalla Headers
02	111301	77	581	2
BinningKKSort	111366	149	581	2
FinalIntegrateFunctor	111309	104	581	2
ForceLJNeigh_for	111360	182	581	2
ForceLJNeigh_reduce	111390	200	581	2
InitialIntegrateFunctor	111319	142	581	2
init_system_get_n	111386	190	581	2
KinE	111293	92	581	2
Temperature	111294	93	581	2
team_policy	111334	150	581	2
nstream	111292	71	581	2
archiver	46331	26075	270	192
capitalize	36534	15378	234	83
condense	33057	1141	227	30
3calibration	82454	31616	351	211
drawing	79523	21825	344	174
laplace	79477	30299	345	200
chat_server	170936	47249	2114	243

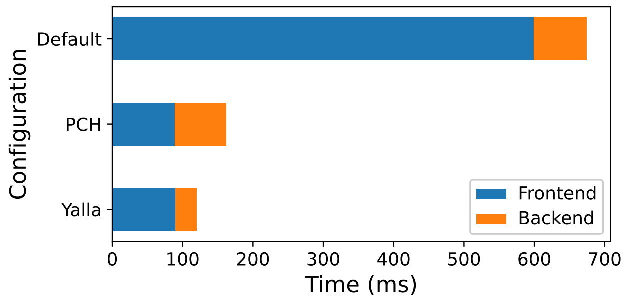
OpenMP backend in PyKokkos to run the contained kernels. We selected the remaining libraries by first finding the most starred C++ repositories that are libraries, and then from those repositories, we selected examples using those libraries as our test subjects. From RapidJSON, we used archiver, capitalize, and condense. From OpenCV, we used 3calibration, drawing, and laplace. From Boost.Asio, we used chat_server. Each of these examples include headers that are expensive to compile from their respective libraries.

5.3 Compilation Time Speedup

We first examine the speedup in compilation time due to Yalla. Table 2 shows compilation times for all test subjects. The first column shows the name of the source file and the second column shows the subject which the source file came from. Columns three through five show the compilation time of the source file (step ④ in Figure 6) using the default, PCH, and Yalla configurations, respectively. The last two columns show the speedup of PCH and Yalla respectively over the



(a) 02 subject.



(b) drawing subject.

Figure 7. Time taken in different Clang compilation phases for two representative subjects.

default. Table 3 shows the statistics of the code being compiled. The first column shows the name of the source file. The second and third columns show the total lines of code (LOC) compiled with the default and Yalla configurations, respectively. The fourth and fifth columns show the total number of headers included (directly and transitively) with the default and Yalla configurations, respectively. To accurately measure the impact of each library on the compilation time of our subjects, we substitute only the library header files in the source files, even if they contain other includes. For example, in the PyKokkos subjects, we substitute only `Kokkos_Core.hpp` and not other header files.

Looking at Table 2, the results show that Yalla speeds up compilation time by 24.5× on average (31.4× for GCC) while PCH speeds up compilation time by 2.8× on average (2.7× for GCC). The PyKokkos test subjects obtain the highest speedups with Yalla, 35.1× on average compared to 3.2× with PCH. We also observe speedups with Yalla of 11.9×, 3.7×, and 9.5× for RapidJSON, OpenCV, and Boost.Asio respectively, compared to 1.2×, 3.1×, and 1.4×.

To understand why the configurations perform differently, we look at Figure 7, which shows a detailed breakdown of the time the compiler spends in the frontend (i.e., Clang lexing, parsing, and semantic analysis) and backend phases (i.e., LLVM optimizations and code generation) for our 02 and drawing test subjects. For the 02 subject (Figure 7a), we observe that PCH greatly reduces the frontend time as the pre-compiled header already has its AST generated. However, the AST must still be loaded from the PCH file on disk which is expensive. Furthermore, the frontend must still perform the required template instantiations in the source file, as it cannot be done without looking at the template usages. The backend time is identical to the default configuration since PCH only improves the frontend time by generating the AST.

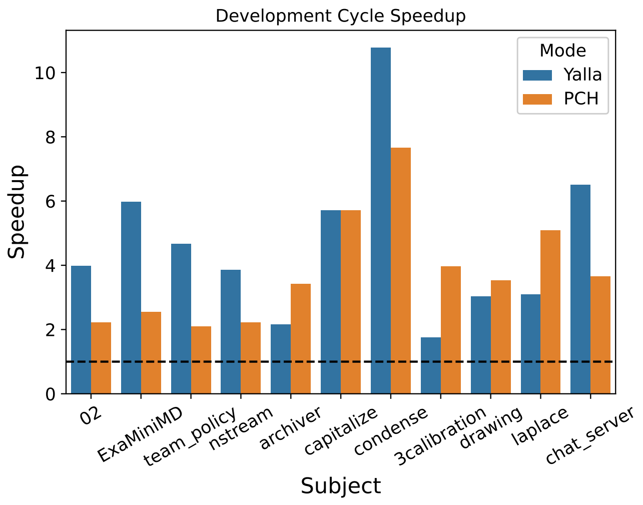


Figure 8. End to end development cycle speedup with Yalla and PCH, including compilation and execution time.

Meanwhile, Yalla provides a larger speedup due to large reductions in both frontend and backend times. To understand why the frontend time is much smaller, we look at the reduction in LOC in Table 3. Yalla reduces the LOC from 111301 to 77 by substituting `Kokkos_Core.hpp`, an expensive header file which pulls in 581 headers in total, including other Kokkos internal headers and standard library headers. Most of these are not used directly in the subject and are therefore removed by Yalla, which keeps only two headers, the lightweight wrappers header and `functor.hpp`. Consequently, the backend time is greatly reduced as well since there is less code to optimize and generate.

We observe a relatively smaller speedup for drawing with Yalla. Figure 7b shows that the improvement in frontend time for Yalla is relatively smaller. Looking at Table 3, we can see that after substituting the expensive headers, Yalla was unable to reduce the total number of header files and LOCs compiled as much as the 02 subject, as drawing directly includes and uses header files other than OpenCV's.

This comparison shows that Yalla provides the largest speedups and improves on PCH for libraries with heavyweight headers (e.g., header-only libraries like Kokkos and Boost.Asio), which commonly occur in the C++ ecosystem [12].

5.4 Development Cycle Improvement

We now look at a scenario where Yalla is used as part of the development cycle: a developer wrote an initial version

```
1   void operator()(int j, int &acc)
2     const {
3    int temp = 0;
4    int i = 0;
5    for (i = 0; i < this->M; i++) {
6     temp += A(j, i) * x(i);
7    }
8
9    acc += y(j) * temp;
10  }
```

(a) C++ Kernel.

```
...
c0: mov  24(%rbp,%rsi,8), %r2
c6: mov  16(%rbp,%rsi,8), %r3
cc: mul  (%rbx,%rsi,8), %r2
d1: mul  (%rbx,%rsi,8), %r3
d7: add  %r1, %r2
db: mov  8(%rbp,%rsi,8), %r4
e1: mul  (%rbx,%rsi,8), %r4
e7: add  %r2, %r3
...
```

(b) x86 Assembly (Default).

```
...
70: movq  48(%rbx), %rdi
74: movl  %r15d, %esi
77: movl  %ebp, %edx
79: callq  0xfd70 <_Z14paren_operator>
7e: mov  (%rax), %r0
82: mov  %r0, 16(%rsp)
88: movq  40(%rbx), %rdi
8c: movl  %ebp, %esi
8e: callq  0xf5f0 <_Z14paren_operator>
...
```

(c) x86 Assembly (Yalla).

Figure 9. The 02 PyKokkos kernel in (a) C++ form, (b) x86 assembly, and (c) x86 assembly after applying Yalla.

of their code and ran it once, following steps ① through ⑤ from Figure 6, made some modifications to their code, and proceeded with the remaining steps in the development cycle, recompile and rerun. This includes steps ④ and ⑤ and the time to rerun the executable. We used relatively small input sizes while measuring the execution time of our subjects after compilation, as we believe this mirrors how a developer would behave when rapidly iterating over their code.

Figure 8 shows speedups provided by Yalla and PCH during the development cycle. The x-axis shows the test subject and the y-axis shows the speedup over the default.

We observe that Yalla and PCH both provide significant speedups over the default configuration in all cases. Compared to each other, Yalla is faster than PCH for all PyKokkos subjects, as well as condense and archiver. In the remaining subjects, the PCH configuration is equivalent to or faster than Yalla. While Yalla is able to provide faster compilation than PCH in almost all cases (see Table 2), this does not always translate to improved development cycle times. First, Yalla requires an additional linking step with the wrappers. Second, the code generated by Yalla is less optimized than the default configuration due to the separation of the class and function definitions from the usages, which restricts the compiler's ability to optimize, as well as due to the introduction of wrappers and dynamic memory allocation. This could lead to larger running times which would slow down the development cycle.

Figure 9 shows how Yalla affects the compiler's ability to optimize our 02 PyKokkos test subject. Figure 9a shows the 02 C++ kernel code generated by PyKokkos, Figure 9b shows the corresponding x86 assembly of the for loop (9b), and Figure 9c shows the x86 assembly after applying Yalla.

The 02 PyKokkos kernel performs a matrix weighted inner product. It accesses two views in every iteration of a for loop (line 6). Indexing views is a call to an overloaded `operator()` method. In Figure 9b, which is the compiler output with all optimizations enabled, these methods are inlined and we see direct memory accesses instead of function calls. In Figure 9c, we see calls to the `paren_operator` method wrapper.

The call instructions do not appear in Figure 9b as the compiler inlines them, i.e., it replaces them with the contents of the function body itself. In Figure 9c, the compiler cannot do so as `paren_operator` is defined in a different translation unit (`wrappers.cpp`).

The extra calls are detrimental to performance for multiple reasons: first, they add performance overhead by creating new stack frames for the called functions; second, inlining the function body at the call site provides the compiler with more context, enabling it to apply additional optimizations that would not have been possible otherwise.

The solution is to inline functions *across* translation units using link-time optimization (LTO) [14], a compiler technique that applies optimizations such as function inlining across a whole program during the linking phase. We experimented with LTO and found that it is able to generate code with the functions inlined, which would look identical to the code in 9b. However, the additional time needed by the linker to apply these optimizations proved to be detrimental to the development cycle, so we did not pursue this further.

5.5 Yalla Startup Cost

We now look at the startup cost of Yalla by measuring its execution time. Note that this is only relevant during the first compilation when Header Substitution takes place.

Figure 10 shows total time taken for initial compilation of the 02 subject (we get similar results for other subjects). The x-axis shows the configuration (Default or Yalla) and the y-axis shows the time in seconds. We show compilation time of the main source file (from which we substitute the header), the tool's execution time, and the compilation time of the function and method wrappers when using Yalla.

The results show that the extra time needed by Yalla is around 2 s, with around 1.5 s for Header Substitution and 0.5 s for wrapper compilation. As Yalla only runs initially, this does not affect the development cycle. Yalla only reruns when a library header is modified or the set of used features from the header changes, which rarely occurs when a developer rapidly iterates on their code. Additionally, this

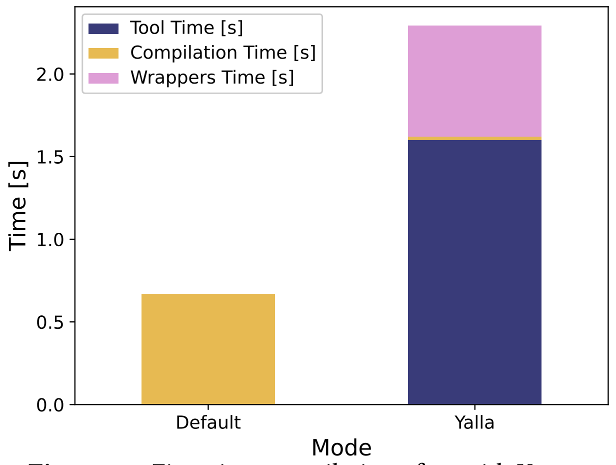


Figure 10. First-time compilation of 02 with Yalla.

implementation of Yalla is a prototype focusing more on correctness than speed, so its execution time can be improved.

6 Future Work

In the future, we plan to apply Header Substitution to entire projects, automatically dividing definitions and declarations across all files to further improve compilation times.

Yalla must be rerun if the set of used symbols from the header file being substituted changes. We will address this by allowing developers to specify all the classes and functions they need prior to running Yalla for the first time.

Header Substitution does not currently support forward declaring nested classes when the parent class must also be forward declared. In the future, we will introduce code transformations that extract nested classes from their parents to allow them to be forward declared.

A drawback of PCH is that the generated files are quite large (in the hundreds of megabytes for our test subjects). Yalla is orthogonal in its approach to PCH so the two techniques can be used simultaneously. We will study the possibility of integrating Yalla with PCH to further improve the development cycle. Additionally, as C++ modules [9] mature, we will study how Yalla could be integrated with them.

7 Related Work

Egalito [33], Zipr++ [26], and BinRec [19] are recompilers that analyze and modify binaries. In theory they can be used to recompile our test subjects; however, this requires compiling source files to map the changes in code down to the binary level which limits what architectures they support.

Copy-and-patch compilation [34] improves run-time compilation speeds by mapping logic to pre-compiled code snippets which are patched in through a run-time system. A custom AST is needed to implement it on a minimal high-level language. Header substitution is more general since it works for any C++ code. ClangJIT [25] is a just-in-time C++ compiler that instantiates templates at runtime to reduce compilation time. Our experimentation with ClangJIT showed that it does not support all of C++'s features.

Include What You Use [7] is a Clang-based tool that detects and removes unused header files. Dayani-Fard et al. [22] presented an approach to detect dependencies at a finer level by removing unused classes and functions from translation units. Their approach is meant to help evolve legacy C++ codebases with bloated header files by making the headers more lightweight. ABC [36] analyzes code changes in headers to speed up incremental builds by skipping unnecessary compilations of unaffected files. Header Substitution speeds up the development cycle by modifying source files to allow classes and functions to be made forward declarable. Molly [21] is a build system with lazy retrieval for Java projects. It only retrieves necessary files for a build instead of the entire library (i.e., the whole jar file), and it integrates code into Java processes at runtime.

Pre-compilation [27, 35] improves build times of header files that rarely change. Our evaluation of PCH and Yalla showed that Yalla can produce code that compiles faster and leads to a faster development cycle most of the time.

Several tools for improving C++ compilation speeds exist. Ccache [4] caches previous compilations to detect if the same compilation reoccurs. *c*Hash [23] hashes ASTs rather than just preprocessed source code, allowing to avoid unnecessary recompilations with more precision. Distcc [6] distributes builds across machines on a network. These tools do not fit our use case of speeding up the local development cycle.

8 Conclusion

We presented Header Substitution, a technique for improving C++ compilation times by eliminating header include statements. Fast compilation time is crucial for a faster development cycle, especially in C++, which suffers from relatively long compilation times. Header Substitution replaces costly header includes in C++ with lightweight headers and extra generated code to enable full access to a substituted header's API. We implemented Header Substitution in a tool named Yalla, showing that it speeds up compilation time by 24.5× on average and development cycle time by 4.68× on average.

Acknowledgments

We thank Cheng Ding, Ivan Grigorik, Tong-Nong Lin, Yu Liu, Hannan Naeem, Pengyu Nie, Aditya Thimmaiah, Zhiqiang Zang, Jiyang Zhang, Linghan Zhong, and the anonymous reviewers for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036, as well as the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003969.

A Artifact Appendix

A.1 Abstract

Our artifact provides the source code of YALLA and our test subjects, along with scripts that reproduce the results in our evaluation section on Ubuntu 20.04 (the version we used) or newer. The script can also be run through Docker. We use a pre-built Clang 15.0.6 x86-64 binary provided by the llvm-project on GitHub.

The source code of YALLA is provided in our own fork of Clang/LLVM here and is cloned and compiled by the provided script. The artifact itself is available at https://github.com/EngineeringSoftware/yalla.

A.2 Artifact check-list (meta-information)

The following is a summary of the contents of this artifact.

- **Algorithm:** Header Substitution.
- **Program:** Yalla and test subjects from PyKokkos, RapidJSON, OpenCV, Boost.Asio.
- **Compilation:** Clang 15.0.6.
- **Transformations:** Header Substitution implemented as a Clang tool.
- **Binary:** We include a binary for Clang 15.0.6, which is the compiler we use to measure compilation times.
- **Run-time environment:** The provided binary is for Ubuntu Linux 20.04 (or newer) and x86-64. We also provide a Dockerfile for containerized execution.
- **Hardware:** Any x86-64 CPU will work. Other CPUs require modifying the script to use a compatible Clang binary.
- **Output:** CSV and JSON files that contain compilation and running times of our test subjects.
- **Experiments:** Implemented through provided bash scripts.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How delivered. A GitHub repository https://github.com/EngineeringSoftware/yalla that contains directions on how to run experiments. The source code of the tool itself and the test subjects will be cloned by the contained scripts.

The hardware and software dependencies listed here are for running the provided script. We provide all source code so switching to other hardware architectures, operating systems, or compilers is possible if the script is modified to use the appropriate binary.

A.3.2 Hardware dependencies. The provided script use a pre-built Clang binary that requires an x86-64 CPU. The script must be modified to obtain a different build in order to run on a different architecture. These builds can be found on LLVM's GitHub Releases page at https://github.com/llvm/llvm-project/releases.

A.3.3 Software dependencies. The Clang binary we use assumes Ubuntu Linux 20.04 and newer. For other operating systems, the script must be modified to obtain a different build of Clang. Other C++ compilers can also be used by modifying the script.

Additionally, we require CMake, git, and Ninja (the Makefile substitute). All can be easily obtained through package managers.

A.4 Installation

To obtain our artifact from GitHub, run

```
git clone https://github.com/EngineeringSoftware/yalla.git
```

A.5 Experiment workflow

Two commands are needed to obtain the experimental results: the first sets up the environment and the second runs the experiments. From the base directory, run `./run.sh setup_all` to obtain all the source code, set up the required compiler, configure a conda environment, and install PyKokkos. Afterwards, run `./run.sh run_all` to run the experiments in the paper.

The experiments can also be run inside a Docker container. To build the image, run `docker build -t yalla-cgo-ae .`, to execute inside the container, run `docker run -it yalla-cgo-ae:latest /bin/bash` for interactive mode, then running the experiments is the same as above. It can also be run in detached mode with `docker run -d yalla-cgo-ae:latest ./run.sh <subcommand>`.

A.6 Evaluation and expected result

All results can be found in the `results/` directory. Compilation times are in CSV files starting with `compilation`. There is a separate CSV file per mode, i.e., normal, PCH, and Yalla, and there are two sets of files, one for PyKokkos subjects (`compilation_kokkos`) and one for benchmarks (`compilation_other`). Running times for PyKokkos subjects are in the `kernels` CSV files for individual kernels and `total` CSV files for the total Python application running time, which is part of the development cycle time. Running times for other subjects are in the compilation CSV files. The `stats` CSV files show LOC and header file statistics.

The results in tables 2 and 3 can be found in the compilation time CSVs and the stats CSV respectively. The development cycle times and speedups in Figure 8 can be obtained by adding the compilation, linking, and running time. Compilation trace results are under `results/traces` in JSON files starting with the subject name and ending with compilation mode (e.g., `02-yalla.json` is for the `02` subject compiled with YALLA configuration). These files can be visualized through Chrome's Trace Viewer (chrome://tracing/) which shows frontend and backend times for each compilation.

A.7 Experiment customization

The experiments can be run with a different compiler, e.g., g++, by modifying the script `run.sh` to use that compiler instead.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] 1998. Boost.Asio. https://www.boost.org/doc/libs/1_86_0/doc/html/boost_asio.html.

[2] 2000. OpenCV. https://github.com/opencv/opencv.

[3] 2014. RapidJSON. https://github.com/Tencent/rapidjson.

[4] 2023. CCache - a fast C/C++ compiler cache. https://ccache.dev/.

[5] 2023. Compiler support for C++20. https://en.cppreference.com/w/cpp/compiler_support/20.

[6] 2023. distcc: a fast, free distributed C/C++ compiler. https://www.distcc.org/.

[7] 2023. Include What You Use. https://github.com/include-what-you-use/include-what-you-use.

[8] 2023. Lambda expressions. https://en.cppreference.com/w/cpp/language/lambda.

[9] 2023. Modules. https://en.cppreference.com/w/cpp/language/modules.

[10] 2023. Templates. https://en.cppreference.com/w/cpp/language/templates.

[11] 2023. Using Precompiled Headers. https://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html.

[12] 2024. Awesome HPP. https://github.com/p-ranav/awesome-hpp.

[13] 2024. Class Declaration. https://en.cppreference.com/w/cpp/language/class.

[14] 2024. LLVM Link Time Optimization: Design and Implementation. https://llvm.org/docs/LinkTimeOptimization.html.

[15] 2024. Matching the Clang AST. https://clang.llvm.org/docs/LibASTMatchers.html.

[16] Nader Al Awar, Neil Mehta, Steven Zhu, George Biros, and Milos Gligoric. 2022. PyKokkos: Performance Portable Kernels in Python. In *International Conference on Software Engineering: Companion Proceedings*. 164–167. https://doi.org/10.1145/3510454.3516827

[17] Nader Al Awar, Steven Zhu, George Biros, and Milos Gligoric. 2021. A Performance Portability Framework for Python. In *International Conference on Supercomputing*. 467–478. https://doi.org/10.1145/3447818.3460376

[18] Abdulaziz Alaboudi and Thomas D. LaToza. 2021. Edit - Run Behavior in Programming and Debugging. In *Symposium on Visual Languages and Human-Centric Computing*. 1–10. https://doi.org/10.1109/VL/HCC51201.2021.9576170

[19] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, Herbert Bos, and Michael Franz. 2020. BinRec: Dynamic Binary Lifting and Recompilation. In *European Conference on Computer Systems*. 1–16. https://doi.org/10.1145/3342195.3387550

[20] Marcel Böhme, Ezekiel O. Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the Bug and How is It Fixed? An Experiment with Practitioners. In *Foundations of Software Engineering*. 117–128. https://doi.org/10.1145/3106237.3106255

[21] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for Java projects. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 643–654. https://doi.org/10.1145/2950290.2950358

[22] Homayoun Dayani-Fard, Yijun Yu, John Mylopoulos, and Periklis Andritsos. 2005. Improving the Build Architecture of Legacy C/C++ Software Systems. In *Fundamental Approaches to Software Engineering*. 96–110. https://doi.org/10.1007/978-3-540-31984-9_8

[23] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler, and Daniel Lohmann. 2017. cHash: Detection of redundant compilations via AST hashing. In *USENIX Annual Technical Conference*. 527–538. https://doi.org/10.5555/3154690.3154740

[24] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003

[25] Hal Finkel, David Poliakoff, Jean-Sylvain Camier, and David F. Richards. 2019. ClangJIT: Enhancing C++ with Just-in-Time Compilation. In *International Workshop on Performance, Portability and Productivity in HPC*. 82–95. https://doi.org/10.1109/P3HPC49587.2019.00013

[26] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. 2017. Zipr++: Exceptional Binary Rewriting. In *Workshop on Forming an Ecosystem Around Software Transformation*. 9–15. https://doi.org/10.1145/3141235.3141240

[27] Tara Krishnaswamy. 2000. Automatic Precompiled Headers: Speeding up C++ Application Build Times. In *Workshop on Industrial Experiences with Systems Software*. 57–66. https://doi.org/10.5555/1251503.1251510

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *International Symposium on Code Generation and Optimization*. 75–88. https://doi.org/10.1109/CGO.2004.1281665

[29] Dennis Mivelli. 2022. Analyzing and Reducing Compilation Times for C++ Programs.

[30] Benjamin Siegmund, Michael Perscheid, Marcel Taeumel, and Robert Hirschfeld. 2014. Studying the Advancement in Debugging Practice of Professional Software Developers. In *International Symposium on Software Reliability Engineering Workshops*. 269–274. https://doi.org/10.1109/ISSREW.2014.36

[31] Bjarne Stroustrup. 1995. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., USA.

[32] Christian Trott, Luc Berger-Vergiat, David Poliakoff, Sivasankaran Rajamanickam, Damien Lebrun-Grandie, Jonathan Madsen, Nader Al Awar, Milos Gligoric, Galen Shipman, and Geoff Womeldorff. 2021. The Kokkos EcoSystem: Comprehensive Performance Portability for High Performance Computing. *Computing in Science Engineering* 23, 5 (2021), 10–18. https://doi.org/10.1109/MCSE.2021.3098509

[33] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 133–147. https://doi.org/10.1145/3373376.3378470

[34] Haoran Xu and Fredrik Kjolstad. 2021. Copy-and-Patch Compilation: A Fast Compilation Algorithm for High-Level Languages and Bytecode. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485513

[35] Yijun Yu, Homayoun Dayani-Fard, John Mylopoulos, and Periklis Andritsos. 2005. Reducing build time through precompilations for evolving large software. In *International Conference on Software Maintenance*. 59–68. https://doi.org/10.1109/ICSM.2005.73

[36] Ying Zhang, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Ping Yu. 2015. ABC: Accelerated Building of C/C++ Projects. In *Asia-Pacific Software Engineering Conference*. 182–189. https://doi.org/10.1109/APSEC.2015.27

Received 2024-09-12; accepted 2024-11-04