Routing at Compile Time

Chun-Xun Lin*, Tsung-Wei Huang†, Martin D. F. Wong‡
Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, IL, USA
E-mail: {*clin99, †thuang19, ‡mdfwong}@illinois.edu

***Abstract*—The rapid evolution of modern C++ programming language has completely changed the way developers write high-performance and robust applications. By modern, we mean C++17, which has revolutionized the "old-fashion" C++98 in many aspects such as meta-programming, concurrency controls, and functional programming. Despite the tremendous progress in language innovation, research on how these advanced features can improve EDA programs is still nascent. In this paper, we introduce a novel routing framework using the technique of *generalized constant expression* in C++17. Our framework allows a router to take advantage of *compile-time* computation and thus can save a significant amount of engineering effort that would otherwise be issued every time the program runs. By prescribing computation at compile time, the compiler is able to further produce more optimized codes to run faster than ever before. We have evaluated our framework on classic routing problems and have demonstrated promising performance gain over which is done solely at runtime. Our framework has the potential to change many fundamental EDA building blocks and thus can achieve better tool performance and engineering productivity.**

I. Introduction

Grid-based maze routing is a fundamental problem in electronic design automation (EDA) [1] and is a integral part of many routing applications such as global routing [2], [3] and detailed routing [6]. A router is not only a standalone tool in the EDA flow but it also works closely with other tools in various stages to deliver useful information for optimizing wire connection. A high-quality router is definitely positive to improve runtime bottleneck and tool scalability, especially for modern circuit designs which are far more dense and complex than last decades. As a result, the goal of this paper is to revisit the routing problem from a new angle of software engineering using powerful language features of modern C++17.

There has been a great deal of research work on routing algorithms [4], [5], [7]. Prior works are categorized to either breadth-first search (BFS)-based solutions which highly rely on memory to propagate the search space, or depth-first search (DFS)-based variants which typically trade memory or optimality for speedup. While these algorithms have their own pros and cons and most have been extensively applied to real designs over decades, a fundamental assumption they made is *computation at runtime*. In other words, the computation or *algorithm* will not be issued until executables are compiled and loaded to operation system (OS)'s virtual space. This design philosophy has been around for years due to the excellent performance of C++ language since a major release in 1998. While this "old-fashioned" C++98 has been present in existing routers and other tools, modern C++17 is quickly changing the way people develop robust and high-performance applications. Given the unique attributes of routing problems, we have observed a rich set of advanced language features in C++17 can be applied to revolutionize the framework people used to develop routers. To speak in specifics, we are interested in *generalized constant expression*, which enables the compiler to evaluate function values at compile time.

Moving computations to compiler has many benefits in dealing with routing problems. First, developers can save a significant amount of engineering effort that would otherwise be issued every time an executable is loaded. Monotonous routines such as parsing and preprocessing prerequisite libraries cost additional runtime. This can be done at one time with compile-time computation. Second, by prescribing computation at compile time, the compiler is able to further produce more optimized codes to run faster than ever before. The computations are replaced by the results in the executable and the runtime is reduced by consulting the results during execution. Last but not least, defining value expression at compile time makes the project less dependent on third-party libraries or external APIs. Some routers [8], [9] use features known in advance to speed up the routing and those features are typically stored as an isolated library to be loaded during execution time. This dependency can degrade the portability of the router, making integration difficult when operations or file formats need to be redefined.

In this paper, we present a compile-time routing framework based on generalized constant expression of modern C++17. In contrast to existing routing frameworks where computations are issued at runtime, our approach can generate routing solutions at compile time. By writing compiler-friendly meta codes, our framework is advantageous in type safety, auto deduction, and versatility. This enables fine-grained optimization to generate more efficient codes which can accelerate the router to the next level. To the best knowledge of the authors, this is the first work that introduces a compiler-based approach to solve the routing problems. Our contributions are summarized as follows:

- **Routing from compilation**. In contrast to the normal routing methods which are done at runtime time, our framework of routing from compilation introduces a new way to design high performance routers. We have successfully demonstrated the viability of routing at compile time. Our idea can inspire developers to rethink the way they used to

[1]C.-X Lin and T.-W Huang contributed equally to this work.

apply in EDA tools, and to incorporate modern C++ features to broaden the performance gain.

- **Efficient engineering turnaround**. Our framework is advantageous in improving the process efficiency of integrating routers with other tools such as routability-driven placement and timing. Embedding routing solutions into the object codes of executables can greatly facilitate the engineering turnaround, for example, debugging, testing, and runtime crash that require frequent program relaunches to backtrace the bugs and faults.
- **Unified framework**. The proposed framework makes a router more standalone, portable and easier to integrate with other tools. By pushing computations that are known beforehand or by algorithms to compiler, our runtime suffers from less dependency on external libraries and is more flexible when the software advances to the next generation.

Experimental results have demonstrated routing at compile-time can save considerable execution time, which has the potential to change the way people develop the tools. With integrating the routing at compile-time, the turnaround time can also be greatly reduced by avoiding recomputing in each iteration when testing or debugging the tools.

The rest of the paper is organized as follows. In section II, we formulate the incremental grid-based routing problem. In section III, we introduce two C++ new features, generalized constant expression and template meta-programming, which enable fine-grained compile-time computation. In section IV, we first illustrate the routing algorithm and analyze the pros and cons of compile-time routing using the two features, then a compile-time router framework with the desirable feature is present. Our experimental results is in section V and the conclusion in section VI.

II. Problem Formulation

In this paper, we consider the grid-based maze routing problem under the presence of weights. Weighted maze routing plays an fundamental role in many routers and other EDA tools. The input is a two-dimensional grid graph $G = \{V, E, W, N\}$, where V is the node set for the cells, E is the edge set for neighboring connections, W is the weight vector denoting the cost of vertices or edges, N is a set of two-pin nets with each net consisting of a source node and a target node. For each source-target pair, a valid route is a path starting from a source vertex and ends at the corresponding target vertex. The length of a route is the sum of all weights in the route. The goal is to establish connections for these two-pin nets while minimizing the congestion cost given by inputs. Instead of developing a complete router which requires more sophisticated constraints (e.g., timing, design rule checking), the goal of this paper focus on a prototype of compiler-driven routing framework. We aim to discover the feasibility of moving a common negotiation-based routing procedure to compiler in order to speed up the engineering turnaround that would otherwise be repeatedly invoked during the runtime, for instance, debugging and testing timing programs given a constant routing benchmark. The proposed framework is illustrated in Figure 1. Compared to the regular routing flow at runtime, the input benchmarks are modified as parts of the source codes and will be compiled together to generate solutions in our framework. It is observed that our framework has less overhead from disk IO and dynamic memory managements at runtime.

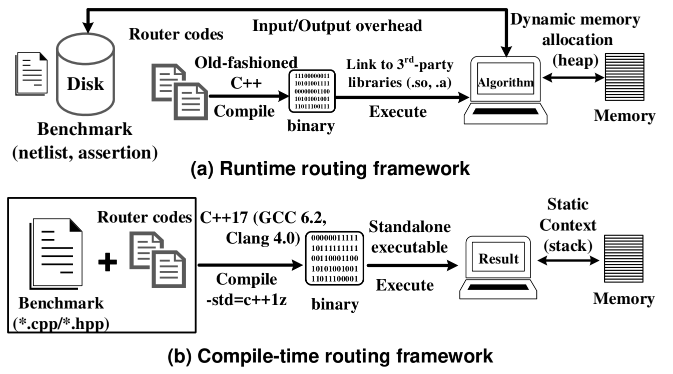

Fig. 1: The proposed compiler-driven routing framework and its comparison to regular routing flow at runtime.

III. Compile-time Computation in C++17

In this section, we introduce essential C++ features that enable compile-time computation. C++ has experienced an enormous advancement in 2017, also referred to as C++17. More and more useful features such as auto type deduction and template programming are added into C++, allowing the programmers to design more efficient software. With the evolution of C++, the compiler has also been updated to support the latest standard. A powerful feature of C++ is the ability to perform compile-time computation, which is to enable the compiler to evaluate the values of objects in compile-time. This occurs when a program uses generalized constant expression [10] or template meta-programming. We shall brief the concept of meta-programming and generalized constant expression and then address the challenges of compile-time computation.

A. Modern Meta-programming

Template programming paradigm is the first C++ feature that enables compile-time computation. Template is developed by C++ in the early 90s and the motivation is to obviate rewriting similar codes for different data types. A template describes the generic way to create an object such as a function or a class, and the compiler instantiates a template when the program invokes the template with a given type. By using templates, programmers can drive the compiler to generate constant data in compile-time. To enhance the functionality of template, modern C++ has introduced *variadic template*. Unlike a normal template where the number of input arguments are fixed, a variadic template can accept variable number of arguments, which further increases the flexibility of template meta-programming. For instance, Listing 1 demonstrates how we alter the weights of the grid graph to perform incremental routing at compile time [2].

```cpp
template <
  typename T, typename M,
  typename N, typename R
>
struct UpdateWeight;

template <
  typename T,
  template <T...> class M,
  template <T...> class N,
  template <T...> class R, T... r
>
struct UpdateWeight<T, M<>, N<>, R<r...>> {
  using type = R<r...>;
};

template <
  typename T,
  template <T...> class M,
  template <T...> class N,
  template <T...> class R
  T... m, T m1, T... n, T n1, T... r
>
struct UpdateWeight<
  T, M<m1, m...>, N<n1, n...>, R<r...>> {
  using type = typename UpdateWeight<
                        T, M<m...>, N<n...>,
                        R<r..., m1+n1>>::type;
};

template <int...>
struct Numbers
{};

template <int... Weights>
struct Grid {
  using elements = Numbers<Weights...>;
};
```

Listing 1: Routing grid graph at compile time.

The struct `Grid` is a variadic template to represent the universe of a weighted grid graph. The `Weights` is a variadic template argument called *parameter pack* that takes arbitrary numbers of edges. The element in a parameter pack can be access by `pack expansion`, which is the ellipsis "`...`" followed by `Weights` in the struct `Grid`. We introduce another variadic template `UpdateWeight` which operates on four template arguments, `T`, `M`, `N`, and `R` to mimic weight update on computing a shortest route. The template argument `T` denotes the data type of the weight, which could be integer, floating points, and other arithmetic types. The template arguments `R` stores the grid graph with weights updated from the `M` and `N`, and the final grid map is forwarded back to another variadic template `Numbers`. Working on these fundamental template building blocks, we are able to establish sophisticated routing procedures at compile time. Listing 2 demonstrates the usage of our templates to reflect the weight update on a 5×6 grid graph from a given route.

```cpp
using weights = Grid<
    1, 2, 1, 6, 1,
    2, 1, 3, 8, 5,
    1, 3, 1, 1, 7,
    6, 8, 1, 1, 9
  >;
using route = Grid<
    1, 1, 0, 0, 0,
    0, 1, 0, 0, 0,
    0, 1, 1, 1, 0,
    0, 0, 0, 1, 0
  >;

using result = UpdateWeight<int,
     grid_graph::elements, route::elements,
     Numbers<>>::type;
```

Listing 2: Update weights (iterative and incremental) on the grid graph at compile time.

Another key feature introduced by C++17 to facilitate the meta-programming is the generalized constant expression. The idea of constant refers to those variables whose values remain fixed through the lifetime of the program. C++17 extends this scope to compile time by allowing users to prescribe function evaluations on constant values using the semantic `constexpr`. A function or variable declared with constant expression implies its value can be *possibly* derived at compile time. When evaluation is not possible, a constant expression falls back to runtime while the compiler is still able to generate more optimized codes. In practice, constant expression can be used combined together with the templates to write efficient meta codes.

```cpp
constexpr int grid_size = 20;
constexpr int S = -2;
constexpr int T = 17;

constexpr int path(
const int P[N],
const int current,
const int count
){
  if( P[current] == S )
    return count;
  else
    return path(P, P[current], count + 1);
}

constexpr int P[grid_size] = {
  1, S, 1, 2, -1,
  0, 1, 6, 7, -1,
  11, 6, 11, -1, -1,
  15, 11, 12, -1, -1
};

constexpr int length = path(P, T, 0);
```

Listing 3: Backtrace the path using const expression at compile time.

[2]All the code listings present in this paper are written in C++17 and are compilable with GCC 6.2 or Clang 4.0 enabling -std=c++1z.

Listing 3 shows an example of recursive constant expression to retrieve a path trace from a grid graph. Starting from the target node, the function `path` recursively counts the number of nodes on the shortest paths stored in the backtrace array `P`. It is observed that the logic of this simple procedure is identical to many of that found in runtime algorithms using dynamic programming. However, in order to issue compile-time computations, all the arguments passed to the function `path` should be declared as constant values during the evaluation. Adding the keyword `constexpr` provides useful hints to the compiler and makes the compilation easier to generate more optimized codes. In this example, we invoke the function `path` by passing a 4×5 grid graph together with a pair of source and target nodes. With a decent modern compiler, the path from the source to the target can be evaluated at compile time. In other words, the function `path` is no longer present in the object codes.

B. Challenges

While constant operations can be hard-coded to persist as immutable objects at runtime, such method is neither flexible nor general when operations have to be redefined or changed in later developments. As we see, the way C++ defines compile-time computation shares similarities with that of hardware description language (HDL) in modeling electronic systems. Loops and iterations are by default extended and *flattened* by the compiler. This property gives rise to critical challenges in writing efficient meta codes to implement routing algorithms. One of the biggest challenges is to correctly implement the flow control with meta codes. Even if template provides programmers a flexible way to carry out compile-time computation, the underlying implementation, though depending on compilers, can introduce significant resource overhead (e.g., memory) due to the "flatness" property. Also, recursion is the only way to operate on variadic templates, as parameter packs rely on pattern matching to realize the control flow. Without carefully designing templates, the instantiation of template objects can easily go too deep to finish the compilation under reasonable resources.

Writing constant expression codes does provide a better alternative to implement flow control than pure template instantiation. However, constant expression is not as general as template instantiation as it requires every expression unit to be immutable. Also, a constant expression cannot modify the state of external objects beyond its scope (e.g., global variables). Functions called from a constant expression should stick with constant expressions as well. In fact, premier compilers and C++ standards restrict constant expression to be only one line. Even though this constraint has been relaxed by modern C++17, many compiler vendors still suffer from a hard limit on the depth a constant expression can join, including recursion, jumping to other subroutines, and so on. Apparently, such limitation can inevitably affect the capability of a function. As a result, programmers need to rethink their algorithms and data structure to suit with these paradigms in order to gain benefits from compile-time computations.

Algorithm 1: Weighted Grid Graph Routing at runtime

```
Input: $S$: source vertices, $T$: target vertices, $W$: weights
       of edges, $V$, $E$: vertices and edges on the grid
Output: $shortestRoutes$
$dist \leftarrow \{\}$;
$prev \leftarrow \{\}$;
$Q \leftarrow \{\}$;
foreach $v \in V$ do
    $dist \leftarrow dist \cup \{0\}$;
    $prev \leftarrow prev \cup \{0\}$;
end
foreach $(s, t) \in (S, T)$ do
    foreach $v \in V$ do
        $dist[v] \leftarrow \inf$;
        $prev[v] \leftarrow -1$;
    end
    $dist[s] \leftarrow 0$;
    $prev[s] \leftarrow -2$;
    $Q \leftarrow \{s\}$;
    while $Q$ not empty do
        $q \leftarrow min(Q)$;
        if $q == t$ then
            break;
        end
        foreach $n \in neighbor(q)$ do
            if $dist[n] > dist[q] + W[E[n, q]]$ then
                $Q \leftarrow Q \cup \{n\}$;
                $dist[n] \leftarrow dist[q] + W[E[n, q]]$
                $prev[n] \leftarrow q$;
            end
        end
    end
    $route \leftarrow backtrace(prev, t)$;
    $updateWeight(W, route)$;
    $shortestRoutes \leftarrow shortestRoutes \cup route$;
end
return $shortestRoutes$;
```

IV. Routing with Meta-programming

In this section, we first introduce the incremental weighted grid graph routing at runtime. Then, we map the steps in the runtime algorithm to our compile-time routing framework by using the C++ features.

A. Weighted Grid Graph Routing at Runtime

Finding the optimal net routing order has been proved to be a difficult task [11], and the general strategy adopted by routers in EDA is to sequentially route each net. As the goal of this work is to minimize congestion instead of finding the optimal routing, our approach is to route each net incrementally following the input order. To reflect the congestion caused by routed nets, the weights of edges on the grid graph must be updated every time a net is routed.

The classic maze routing problem can be seen as a special case of our weighted grid graph routing problem, where the maze routing problem considers only one source-target pair. The typical way to solve the maze routing problem is via breadth-first search (BFS). BFS starts from the source vertex and propagates through neighboring vertices until reaching the target vertex. Backtracing is employed to retrieve the shortest route after arriving at the target vertex. Our idea is to iteratively apply maze route on each source-target pair and increase the weights of edges on the shortest route. We sketch our incremental weighted grid graph routing algorithm in Algorithm 1

The inputs to the algorithm are the vertices and edges on the grid, weights of edges and the source-target pairs. The algorithm finds the shortest route for each source-target pair from line 8 to 32. From line 9 to 27, the maze routing starts propagation from the source vertex and stops when target vertex is found. In line 28, the `backtrace` subroutine starts from the target vertex to retrieve all nodes on the shortest route and another subroutine `UpdateWeight` increases the weights of edges on the route in line 29. In line 30, the shortest route is stored in the output.

B. *Compile-time routing framework*

Our goal is to map the runtime routing algorithm to the compile-time routing framework using the two C++17 features and we demonstrate how to convert the essential operations to compile time.

First, we observe the routing algorithm has count-based loops and a condition-control loop in control flow. The constant expression has no difficulty in realizing these control flows whereas employing recursion of template for the loops is more complicated:

```cpp
template <typename T,
        typename G,
        int Pair,
        int Count> struct ForLoop;
template <typename T,
        template <T...> class G,
        T... g, int Count>
struct ForLoop<T, G<g...>, 1, Count> {
  static const int count = Count;
};

template <typename T,
        template <T...> class G,
        T... g, int Pair, int Count>
struct ForLoop<T, G<g...>, Pair, Count> {
  static const int count =
      ForLoop<T,G<g...>,Pair-1,Count+1>::count;
};
```

Listing 4: A simple for-loop using template.

Listing 4 is a simple example of using a template `ForLoop` to loop `Pair` times. In this example, the struct `ForLoop` repeatedly instantiates itself with decreasing the `Pair` by 1 every time until `Pair` becomes 1.

Next, we need to update the values of variables in compile time as the routing algorithm has mutable data such as the weights and distance. For template, variables cannot be modified after initialization. In other words, changing the values of variables is equivalent to create new variables to hold the values. For constant expression, an important restriction is unable to modify the status of external objects. To cope with the immutability restriction, a workaround is to create a local array with fixed size for the algorithm manipulation, which requires to know the size of data in advance.

The last operation is to access the data in compile time. An important observation is that the data access in routing algorithm is very irregular. During the propagation the update order is determined by the distance between each vertex and the source vertex. Consequently only a subset of vertices might be access and those vertices might vary greatly from pair to pair. For variadic template, to retrieve a specific element in a parameter pack , one way is to recursively visit the elements until it is reached. Listing 5 demonstrates how to access the N_{th} element in a parameter pack.

```cpp
template <typename T, typename P, int N>
    struct GetNthElement;

template <typename T,
          template <T...> class P,
          T p1, T... p2>
struct GetNthElement<T, M<p1,p2...>, 0> {
  static const T value = p1;
};

template <typename T,
          template <T...> class P,
          T p1, T... p2, int N>
struct GetNthElement<T, P<p1,p2...>, N> {
  static const T value =
      GetNthElement<T,P<p2...>,N-1>::value;
};
```

Listing 5: Access an element in parameter pack. (method 1)

The struct `GetNthElement` is a variadic template to recursively search the N_{th} element in the parameter pack `P`. This method is less efficient due to the recursive element visit. Another way is to put all parameters in the array and returns the N_{th} element. Listing 6 shows the second method.

```cpp
template <typename T, typename M, int N>
struct GetNthElement;

template <typename T,
        template <T...> class M,
        T... m, int N>
struct GetNthElement<T,
        M<m...>, N> {
  static constexpr T array[sizeof...(m)] =
      {m...};
  static constexpr T value = array[N];
};
```

Listing 6: Access an element in parameter pack. (method 2)

The parameter pack M is put into `array` and the N_{th} element is extracted in `value`. For constant expression, the access to specific data is the same as access an element in an array.

To summarize, the proposed compile-time routing framework is analogous to the runtime algorithm except some changes are necessary in order to conform to the restriction of compile-time computation. We detail those changes below. First, one of the inputs, weights of edges, has to be updated during routing. To handle this, a temporary array with the same size is created to avoid modifying the input.

Second, unlike the runtime algorithm that can dynamically allocate memory, the size of local arrays has to be determined before compiled. In the routing algorithm, the size of `dist` and `prev` is set to the grid size. For the array `Q` that keeps the frontier vertices, the possible maximum number of frontier vertices is equivalent to the perimeter of the grid and thus the size of `Q` is set to the perimeter of the grid.

Third, the output `shortestRoutes` cannot be an input parameter of the function as `shortestRoutes` will be modified during routing. Therefore, in compile-time routing the `shortestRoutes` is declared as a local array inside the function and will be returned at the end of function. Determine the way to store a shortest route is of critical importance since this affects the size of `shortestRoutes`. For each shortest route, we only store the source vertex, target vertex and bends, which can save considerable space.

V. Experimental Results

We implemented our framework in C++ language and conducted experiment on a machine with a 2.4 GHz CPU and 33 GB memory. Our framework is compiled by Clang 4.0 enabling the flag `-std=c++1z` to support for the latest C++17 standard. We did not select the latest release of GCC 6 because many advanced C++17 features are still under construction. There should be no significant difference between compiler vendors as our framework follows the official C++ standards. We randomly generate 10 test cases of 500×500 grid graphs and run each test case over different numbers of nets. The runtime routing algorithm presented in Algorithm 1 is considered as the baseline. For simplicity, we refer the routing at compile time as the compile-time router and routing at runtime as the runtime router. The elapsed time of each program is denoted as "execution time." We report the execution time of each program on an average of 10 runs. Both programs are compiled with O2 optimization flag to drive compute-optimized codes.

We first compare the average execution time of both frameworks on each benchmark. Figure 2 shows the execution time. It is expected that the compile-time router completes the routing faster than the runtime router over all test cases. The execution time of the runtime router increases linearly with the problem size while the counterpart of the compile-time router remains almost zero. Figure 3 shows the speedup of each set. The compile-time router achieves an order of magnitude speedup over the runtime router in each set and the speedup continues to scale with the increase of problem size.

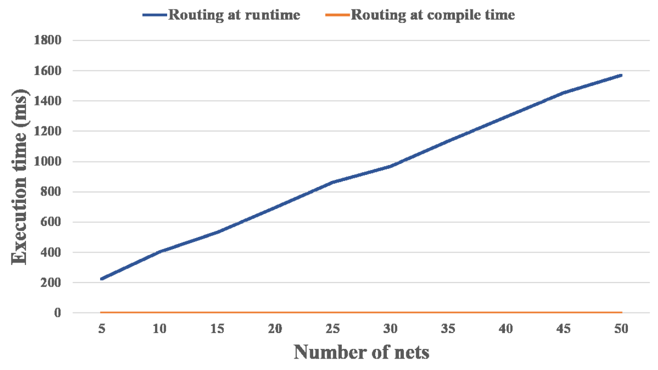

Fig. 2: Comparison between routing at compile time and routing at runtime

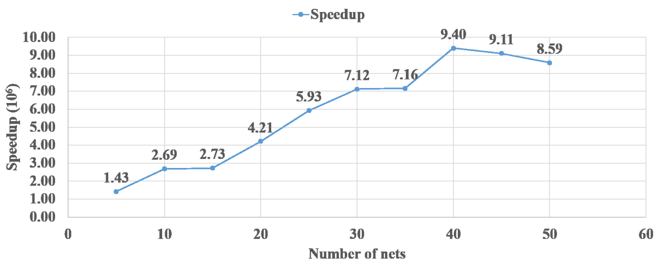

Fig. 3: The speedup of compile-time routing over runtime routing

The benefit of our framework can be clearly observed in this experiment. By pushing computations to compiler, we are able to embed routing solutions into the objects of the executable. For applications where routing blocks are treated as fixed black box, our framework provides another optimization opportunity for further speedup.

Next, in Figure 4 we demonstrate the compilation time of the compile-time router on each benchmark. The compilation time grows linearly with the problem size, which is similar to the execution time of the runtime router. Though expected, a great advantage of the compile-time router is that routing is performed only once from compilation, and the result can be reused many times during the runtime with almost zero computation. This can greatly reduce the time and effort for engineering turnaround. For example, consider a flow where we need to extract the parasitics from a fixed routing block for timing analysis as shown in Figure 6. During the timing, the routing blocks remain unchanged and it is not efficient to reroute the circuits for each turnaround iteration. While it is true the solutions can be pre-stored in the database for later reuse, such methods still involve frequent disk I/O overhead. Instead, our framework provides another alternative to improve this issue by routing from compilation. As the routing solutions are embedded in the compile-time router, substantial turnaround time can be saved by avoiding recomputation in each turnaround iteration, for example, debugging the faults of the timer. Figure 5 shows the potential saving of engineering efforts in terms of runtime improvement by adopting our framework.

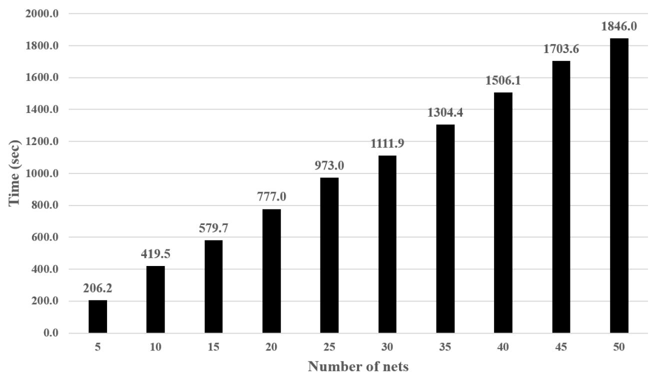

Fig. 4: Compilation time of compile-time router.

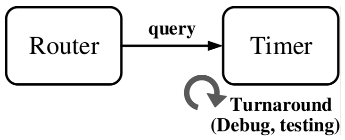

Fig. 5: Turnaround between router and timer.

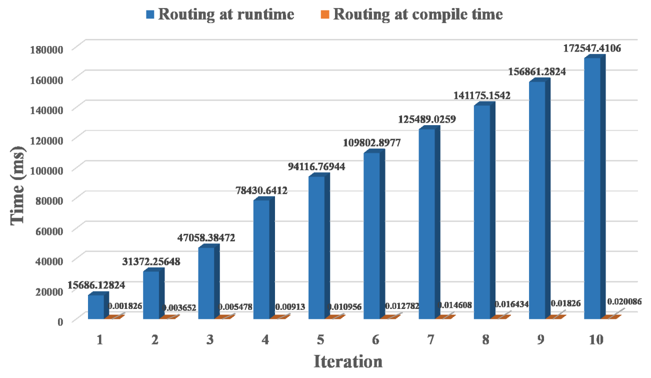

Fig. 6: Turnaround cost for routing at compile time and runtime.

VI. Conclusion

In this paper, we present a routing at compile time framework that can substantially reduce the execution time by prescribing computation to compilation. We utilize the two powerful C++17 features: variadic template and generalized constant expression that enable the computing at compile time to build our routing at compile time framework. The experimental results show that, compared with a routing at runtime method, routing at compile time can achieve an order of magnitude speedup and the execution time remains close to zero even when the problem size increases. We further demonstrate with integrating the compile-time computation, considerable engineering turnaround can be saved by avoiding recomputation. Our work shows the potential of using compile-time computation to design better tools and benefit the EDA flow. For future work, we plan to investigate potential approaches to speed up the compilation, such as employing an existing distributed build tool [12] or designing a distributed compilation flow on a distributed execution engine [13].

VII. Acknowledgment

This work is partially supported by the National Science Foundation under Grant CCF-1421563 and CCF-171883.

References

[1] Lee, Chin Yang, “An algorithm for path connections and its applications,” *IRE transactions on electronic computers*, (3), pp. 346–365, 1961.

[2] Liu, Wen-Hao and Kao, Wei-Chun and Li, Yih-Lang and Chao, Kai-Yuan, “Multi-threaded Collision-aware Global Routing with Bounded-length Maze Routing,” *ACM/IEEE DAC*, pp. 200–205, 2010.

[3] Liu, Wen-Hao and Kao, Wei-Chun and Li, Yih-Lang and Chao, Kai-Yuan, “NCTU-GR 2.0: Multithreaded Collision-Aware Global Routing With Bounded-Length Maze Routing,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 709–722, 2013.

[4] Soukup, J., “Fast Maze Router,” *ACM/IEEE DAC*, pp. 100–102, 1978.

[5] Hadlock, FO, “A shortest path algorithm for grid graphs,” *Networks*, pp. 323–334, 1977.

[6] Zhang, Yanheng and Chu, Chris, “RegularRoute: An Efficient Detailed Router with Regular Routing Patterns,” *ISPD ’11*, pp. 45–52, 2011.

[7] Huang, Tsung-Wei and Wu, Pei-Ci and Wong, Martin D. F, “UI-Route: An Ultra-Fast Incremental Maze Routing Algorithm,” *SLIP ’14*, pp. 4:1–4:8, 2014.

[8] C. Chu and Y. C. Wong, “FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design.” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 70–83, 2008.

[9] G. Ajwani and C. Chu and W. K. Mak, “FOARS: FLUTE Based Obstacle-Avoiding Rectilinear Steiner Tree Construction,” *TCAD ’11*, pp. 194–204, 2011.

[10] Dos Reis, Gabriel and Stroustrup, Bjarne, “General Constant Expressions for System Programming Languages,” *SAC ’10*, pp. 2131–2136, 2010

[11] L. C. Abel, “On the Ordering of Connections for Automatic Wire Routing,” *IEEE Transactions on Computers*, pp. 1227–1233, 1972

[12] distcc: http://distcc.org/

[13] Tsung-Wei Huang, Chun-Xun, Lin, and Martin D. F. Wong, “DtCraft: A Distributed Execution Engine for Compute-intensive Applications,” in *IEEE/ACM ICCAD*, 2017.