Bambu Recurse: Extending Bambu to Synthesize Recursion

Andrew Kim
University of Maryland
U.S.A.

Caspar Popova
University of Maryland
U.S.A.

Abstract

High level synthesis tools allow code written in high level languages to run on hardware. However, many synthesis tools disallow high level abstractions, such as recursion, due to the complexity of hardware support. This restriction limits the expressivity of code that can be compiled to hardware. In this project, we expand the synthesizable subset of an open-source synthesis tool, Bambu, by transforming recursion to iteration.

1 Introduction

High level synthesis tools take software written in high level languages such as C/C++ and automatically convert it into hardware description languages such as Verilog. While high level synthesis (HLS) is very promising in accelerating the process of hardware design by describing hardware algorithmically rather than manually, there remain some limitations. Many HLS tools only support a restricted subset of C/C++ software. This means that many valid programs cannot be included in synthesis due to utilization of language constructs which are difficult or impossible to represent in static hardware. These include recursion, higher order functions, polymorphism, and exception handling. HLS tools often rely on traditional compilers, such as GCC or Clang for C/C++, to eliminate these constructs, therefore limiting optimization opportunities within the HLS backend itself. Extending existing HLS tools to support and optimize high level language constructs in C/C++ can make HLS tools more practical and accessible by expanding the synthesizable subset to more expressive programs.

Recursion is a natural high-level representation for algorithms like traversals, backtracking, and sorting. However, recursion is not supported by many HLS tools, including Bambu, Legup, and Vitis. The major challenge of synthesizing recursion is that recursion relies on dynamic and potentially unbounded resources in terms of the stack. While some high-level programmers may choose to manually rewrite their recursive code to its iterative equivalent such that it can be synthesized, this process is time-consuming, error-prone, and yields complex code than the original recursion. Thus, it is natural to desire recursion support in HLS tools.

In this work, we focus on expanding support for recursion in the HLS tool Bambu [1]. Bambu is an advanced full HLS pipeline to Verilog. It supports common HLS datasets PolyBench, MachSuite, and CHStone and is used in real-world applications like NanoXplore and by the European Space Agency. Unlike Legup and Vitis, Bambu is open source, making it approachable to new functionality. Bambu cannot reliably synthesize recursion, namely non-tail call recursion. Bambu can only synthesize tail-call-recursive programs by relying on tail call optimization implemented in its GCC frontend. Consequently, the expressivity of the input space is constrained for programmers.

Our work Bambu Recurse[1] extends the Bambu tool directly with support for certain forms of non-tail call recursion via recursion removal. This extension may eventually support a wide body of recursive algorithms such as traversals, and dynamic programming.

In the remainder of this paper, we discuss existing work in recursion elimination (Section 2), our approach to recursion removal (Section 3), evaluation (Section 4), and future work (Section 5).

2 Background

There are many approaches to recursion elimination, and consequently external extensions to HLS tools that utilize those approaches to enable recursion support. Liu et. al. describe multiple approaches of what they call recursion-to-iteration optimization, including complex cases such as multiple base cases, multiple recursive calls[3]. These methods are used in HLS tools for recursion elimination.

MLIR-Recursion [2] is a compiler framework based on MLIR designed to support recursion in HLS. Their approach combines two methods for removing recursive code reliant on the stack: transformation into loops and transformation into finite state machines. Their compiler emits standard LLVM and utilizes Bambu as HLS. Unlike MLIR, our project incorporates recursion elimination directly within Bambu, allowing users to skip intermediate tools like MLIR. While outside of our work's scope, recursion elimination within the intermediate representation will allow for more precise optimizations increasing the potential performance improvements for our approach.

In contrast, the FHW project [6], which synthesizes Haskell to Verilog, focuses on HLS for functional languages. FHW's custom back-end to the Glasgow Haskell Compiler (GHC)

[1] https://github.com/cjpopova/bambu-recurse/tree/dev

includes lowering transformations for recursion, polymorphism, and first-order functions that transform these abstractions into a form suitable for hardware. They transform recursive functions into tail-recursive functions that explicitly manipulate the stack. Their transformations are semantics-preserving and can be proven correct. Their recursion transformation requires a continuation passing style (CPS), lambda lifting, and defunctionalization. Unlike FHW, Bambu is designed for C/C++, not a functional language.

HLSRecurse [4] is an embedded DSL in C++ that transforms recursion into state machines and stacks. As a DSL, HLSRecurse requires manual modification to the source code and does not have opportunities for further optimization that require compiler analysis.

While most approaches extend compiler analyses, Xu et. et al utilize LLM-driven program repair of synthesized code[5]. However, this approach has no formal guarantees of correctness and has no opportunities for program optimization.

3 Recursion Removal

3.1 Challenge of Synthesizing Recursion

The major challenge for synthesizing recursion lies in the difficulty determining the resource requirements for hardware implementation. This is because by definition, recursive calls rely on dynamic and potentially unbounded inputs.

```
1 int function(int n) {
2   if(n <= 1) { return 1; }
3   int a = function(n-1);
4   return a * a + n;
5 }
```

Listing 1. Recursive Function

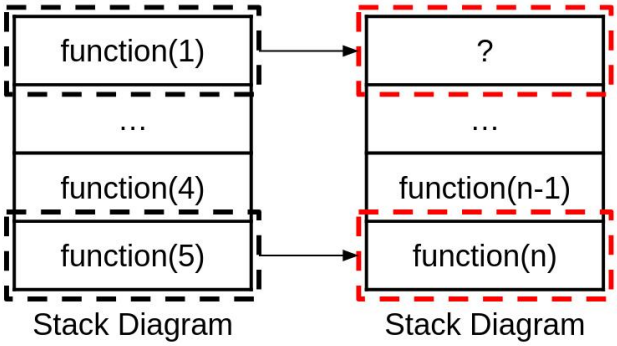

Figure 1. Recursion Stack Memory Problem

The example recursive function in Listing 1 portrays this problem clearly. Depending on the input argument n, the number of stack frames required before hitting the base case will vary as shown in Figure 1. HLS tools such as Bambu therefore will be unable to determine the amount of memory resources required and reject such programs.

```
1 int accumulator(int n, int a=1) {
2   if(n == 0) { return a; }
3   return accumulator(n-1, n*a);
4 }
```

Listing 2. Tail Call Recursive Function

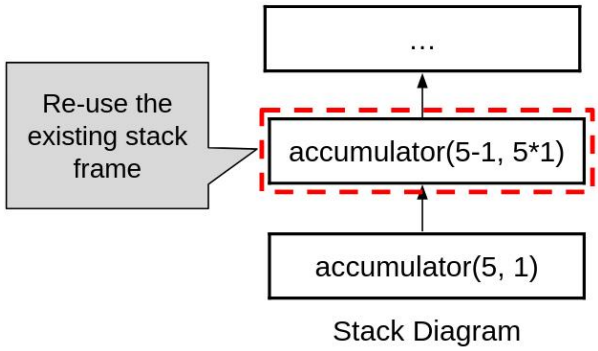

Figure 2. Tail Call Optimization

However, certain recursive functions may be optimized away resulting in a synthesizable program. When the the recursive call is the last statement in the function is a tail call and has no further recursion, as shown in Listing 2, the recursive call may be optimized away. Tail call optimization is a compiler technique where instead of creating a new stack frame and function call, the program instead reuses its own stack frame only modifying any arguments or local variables. This is possible as no further computation is required with the recursive call's return value meaning that a previous state may be discarded. Tail call optimization is oftentimes implemented in a compiler's frontend such as GCC, allowing the recursive function to keep a constant stack frame depth as seen in Figure 2.

Such optimizations nevertheless are not applicable to most practical recursive functions, highlighting a significant limitation for modern HLS systems.

3.2 Approach in Bambu Recurse

We focus on removing a particular type of recursion: single-input recursion. Single input recursive functions, which recursve over a single argument and have a limited structure, like those in Listing 1, can be transformed to efficient loops that use an explicit stack. In this form, the stack is represented by a fixed-length array. Each element of the array represents one stack frame. Pushing a frame onto the stack represents making a recursive call, and popping a frame from the stack represent returning a result.

From a recursive function such as Listing 1, Bambu initially generates an intermediate representation defined in SSA form and basic blocks. Given this intermediate representation, we must identify several key statements and variable

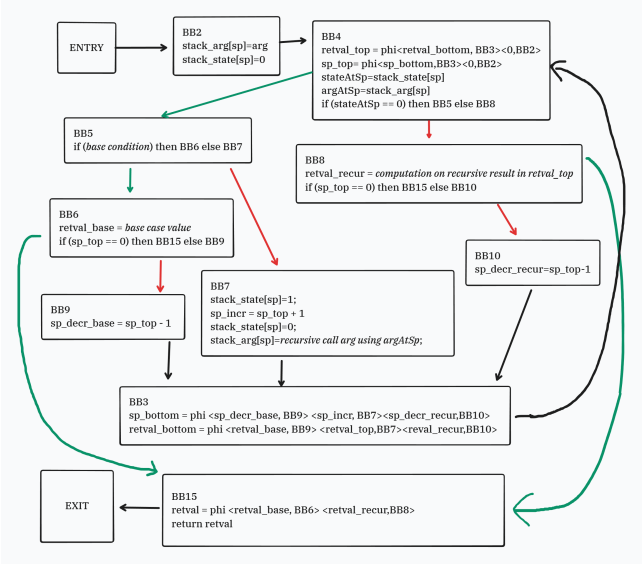

Figure 3. Abstraction of stack-based loop in Bambu IR.

relations to build the stack-based loop. The outline of a generalized stack-based loop is shown in Figure 3. The italicized regions (the base condition, base return value, recursive call argument, and computation on recursive result) specialize the loop to the behavior of the input recursive code. To identify these program semantics, we implemented several program analysis routines.

Base return value and condition. The base case value (`a` in Listing 1) is identified by traversing the AST back from the return value, and identifying which value came from a branch that did not use recursion. The condition (ex: `n <= 1`) is identified by as the branch condition between the base and recursive branches.

Recursive argument and computation. The recursive argument (ex: `n-1`) is identified by the argument passed into the original recursion call. The recursive computation (ex `a * a + n`) is identified as the computation done to the recursive result.

Once essential program semantics have been recovered from the recursive intermediate representation, we can start construction on the new intermediate representation for the iterative implementation. Now, we discuss the components of the stack-based loop shown in Figure 3. The C-like version of the stack-based loop for Listing 1 is shown in Listing 3, which serves as a reference implementation.

Stack. The stack is constructed to simulate the stack frames used in recursion. Like a normal stack frame, our explicit stack holds the function arguments, return value, and any local variables. The stack also hold a state, explained below. The stack pointer (`sp`) points to the index of the current stack frame being executed. During the transformation, a recursive call is replaced with a push operation on the stack. Conversely, a pop operation represents the original return instruction.

```
1  int function_iterative(int n) {
2    unsigned int sp = 0;
3    int stack_arg[512];
4    int stack_state[512];
5    int stack_ret[512];
6
7    stack_arg[sp] = n;
8    stack_state[sp] = 0;
9  loop_start:
10   if(stack_state[sp] == 0) {
11     if(stack_arg[sp] <= 1) {
12       stack_ret[sp] = 1;
13       if(sp == 0) {
14         return stack_ret[0];
15       }
16       sp--;
17       goto loop_start;
18     }
19     stack_state[sp] = 1;
20     sp++;
21     stack_state[sp] = 0;
22     stack_arg[sp] = stack_arg[sp-1] - 1;
23     goto loop_start;
24   }
25   if(stack_state[sp] == 1) {
26     stack_ret[sp] = stack_ret[sp+1] * stack_ret[sp
           +1] + stack_arg[sp];
27     if(sp == 0) {
28       return stack_ret[0];
29     }
30     sp--;
31     goto loop_start;
32   }
33 }
```

Listing 3. Iterative Function

States. States represent the which phase of the recursive call a stack frame is in. In our model, state of 0 represents before, and nonzero state represents after recursive call. Such a marker is necessary to support tail recursion, where computation may occur with the return value. As an example, a state of 0 for Listing 1 would be the base case condition, while a nonzero state would refer to the computation `a * a + n`.

Loop Condition. Since recursive calls are removed, a loop condition is required to determine when the program can return. Defined in basic blocks 3, 4, and 8 of Figure 3, it checks if the state is 0. If so, the current stack frame has not made its recursive call and the program must continue. If the current stack frame is 0 and stack pointer is 0 (no more frames left on the stack), the program is complete and may return. Otherwise there are still stack frames left to process and the program must continue.

These components combined with the program semantics pulled from the recursive intermediate representation, our

model can successfully eliminate recursion. The fully transformed intermediate representation for Listing 1, is shown in Figure 3. Listing 3 depicts the source code for the exact basic block intermediate representation structure.

4 Evaluation

We evaluate the expressivity and performance of our Bambu extension. First, we discuss the expressivity added to the synthesizable subset of C supported by Bambu. Second, we compare the performance and hardware resource benchmarks of recursive programs synthesized with our extension to the performance of their iterative equivalents.

The benchmark set is based on that of the MLIR recursion project[2], which includes the fibonacci and ackerman in their recursive and manually-implemented iterative variants.

4.1 Expressivity

Bambu Recurse extends the Bambu HLS with support to synthesize single-input recursion, a limited form of non-tail call recursion. While limited, this subset still encompasses a large set of programs including fibonacci and ackerman that are now synthesizable. Additionally, this set of programs demonstrates that HLS support for recursion is preferable to manually rewriting. Table 1 compares the lines of code in recursive benchmarks versus their manually written iterative equivalents. Manually rewriting even the simplest benchmarks to iteration requires a 4-5x increase in lines of code, and a jump of complexity.

Benchmark	$\text{LoC}_{\text{recurse}}$	LoC_{iter}
Function	5	19
Ackerman	15	78
Fibonacci	10	57

Table 1. Lines of code in recursive vs iterative benchmarks.

4.2 Performance

Our extension to the Bambu compiler was made to version PandA 2024.10. The frontend used GCC8 at O0, to prevent the limited GCC recursion optimizations that would complicate our analysis. The performance simulation was performed using Verilator 4.038.

Table 2 shows the hardware resources required for both a manually written iterative version and the recursive form of the program. Across all metrics, recursive programs require more hardware resources. Registers ~40% increase, multiplexers ~50-60% increase, flip-flops ~20% increase, and ~60% increase in area. Despite these results, we observed that the hardware resources for the manually written functions could vary quite drastically depending on the implementation of iterative programs.

[2]https://github.com/jiangnan7/MLIR-Recursion/tree/main

Table 3 shows the cycle count of the manually written iterative and original recursive forms of the program. Each pair of benchmarks is simulated on the same representative inputs. Across all benchmarks, synthesized recursive programs require more cycles to complete. Both the increased resource usage and decreased performance of Bambu Recurse can be attributed to lack of optimization in our extension, which is discussed in the next section.

Benchmark	Registers	Multiplexers	Flip-flops	Area
$\text{Function}_{\text{iter}}$	9	4	228	375
$\text{Function}_{\text{recurse}}$	19	13	337	913
$\text{Ackerman}_{\text{iter}}$	23	19	370	871
$\text{Ackerman}_{\text{recurse}}$	35	35	519	1451
$\text{Fibonacci}_{\text{iter}}$	20	12	355	695
$\text{Fibonacci}_{\text{recurse}}$	20	21	305	1114

Table 2. Hardware Resource benchmarks

Benchmark	Recursive	Iterative
Function	176	64
Ackerman	9	7
Fibonacci	833	599

Table 3. Cycle count of recursive vs iterative benchmarks

5 Limitations and Future Work

5.1 Limitations

Throughout the project we faced several key technical challenges. During the open source software installation, we found several inaccuracies requiring a custom environment to handle all of Bambu's dependencies. Additionally, lack of documentation for Bambu's custom intermediate representation, based on GCC's Gimple, consequently resulted in slower prototyping for our explicit stack representation. Finally, designing a generalizable recursion to iteration translation technique required several iterations as well as the implementation of complex program analysis routines.

Due to time constraints and implementation difficulties, our work has several limitations. Because our stack and loop condition infers the stack state as a binary variable, we are only able to support single input recursion. This means that multiple input recursion where a function may make more than one recursive call and complex recursive patterns including polymorphism are not supported.

The negative results in resource usage and performance are attributed to lack of optimization. Unlike MLIR Recursion[2], we did not have access to a library of advanced compiler analyses to enable optimizations. Because Bambu is an independent, open-source tool, we implemented minimal analyses

for implementing recursion removal from scratch. Optimizations could reduce the required stack space by removing unnecessary local variables or caching repeatedly solved subproblems. Such optimizations may have improved performance and reduce the resource overhead on the target hardware. We were unable to compare the performance and resource overhead of our approach to MLIR Recursion due to time and installation constraints.

Finally, the use of an explicit stack with a pre-defined depth to remove recursion is fundamentally limited by the space in the underlying hardware. In our current model, there is no support for a stack overflow. This issue also exists in MLIR Recursion.

5.2 Future Work

While our current recursion removal pass only supports single input recursion, an extension to handle multiple recursive calls would be possible. Such an extension would require a modification to the loop condition which would have to consider an arbitrary number of states as well as more complex program analysis routines to identify different states. Alternative extensions include an optimization pass over the transformed intermediate representation. This would present an opportunity to improve the overall performance of the synthesized hardware.

6 Conclusion

We introduced a recursion removal pass within the Bambu HLS framework which systematically translate recursive functions into iteration through the use of an explicit stack. By combining program analysis routines and mapping them to the stack based loop intermediate representation we are able to support single input recursion. Our work presents an effective method for enhancing the practicality of HLS tools and increasing the overall subset of synthesizable programs.

References

[1] Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, Marco Minutoli, Christian Pilato, and Antonino Tumeo. 2021. Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1327–1330. doi:10.1109/DAC18074.2021.9586110

[2] Jiangnan Li, Zhengyi Zhang, Xuegong Zhou, and Lingli Wang. 2024. An MLIR-Based Compiler for Hardware Acceleration with Recursion Support: (PhD Forum Paper). In *International Conference on Field Programmable Technology, ICFPT 2024, Sydney, Australia, December 10-12, 2024*. IEEE, 1–4. doi:10.1109/ICFPT64416.2024.11113451

[3] Yanhong A. Liu and Scott D. Stoller. 2000. From Recursion to Iteration: What are the Optimizations?. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '00), Boston, Massachusetts, USA, January 22-23, 2000*, Julia L. Lawall (Ed.). ACM, 73–82. doi:10.1145/328690.328700

[4] David B. Thomas. 2016. Synthesisable recursion for C++ HLS tools. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 91–98. doi:10.1109/ASAP.2016.7760777

[5] Kangwei Xu, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, Ulf Schlichtmann, and Bing Li. 2024. Automated C/C++ Program Repair for High-Level Synthesis via Large Language Models. arXiv:2407.03889 [eess.SY] https://arxiv.org/abs/2407.03889

[6] Kuangya Zhai, Richard Townsend, Lianne Lairmore, Martha A. Kim, and Stephen A. Edwards. 2015. Hardware synthesis from a recursive functional language. In *2015 International Conference on Hardware/-Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, Gabriela Nicolescu and Andreas Gerstlauer (Eds.). IEEE, 83–93. doi:10.1109/CODESISSS.2015.7331371

Submitted 20 December 2025