Bambu Recurse

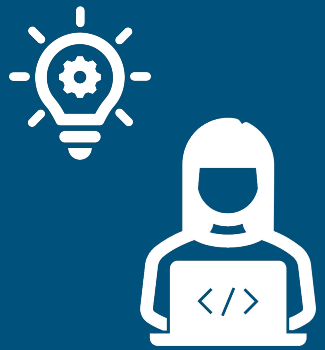
Supporting recursion in Bambu HLS

Andrew Kim & Caspar Popova

Supporting recursion in Bambu HLS

Why support recursion?

- Natural algorithm representation (traversal, backtracking, sort)
- Reduces rewriting effort compared to iterative counterparts
- Clarity, less complex than iterative implementations



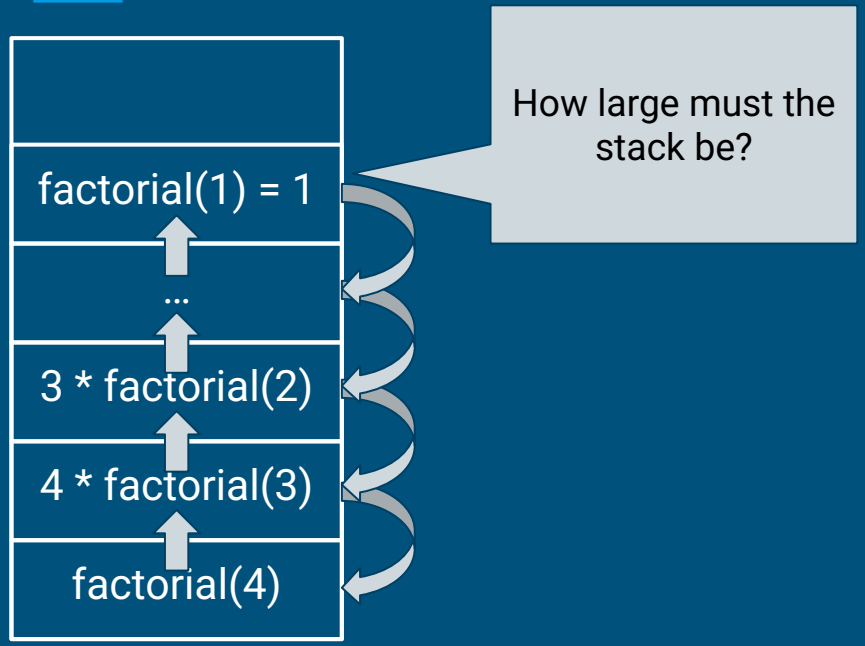
Lack of Recursion Support

HLS tools: Bambu, Legup, Vitis

Bambu HLS:

- Full HLS pipeline
- Open Source
- Proven utility in real world applications (NanoXplore, ESA)

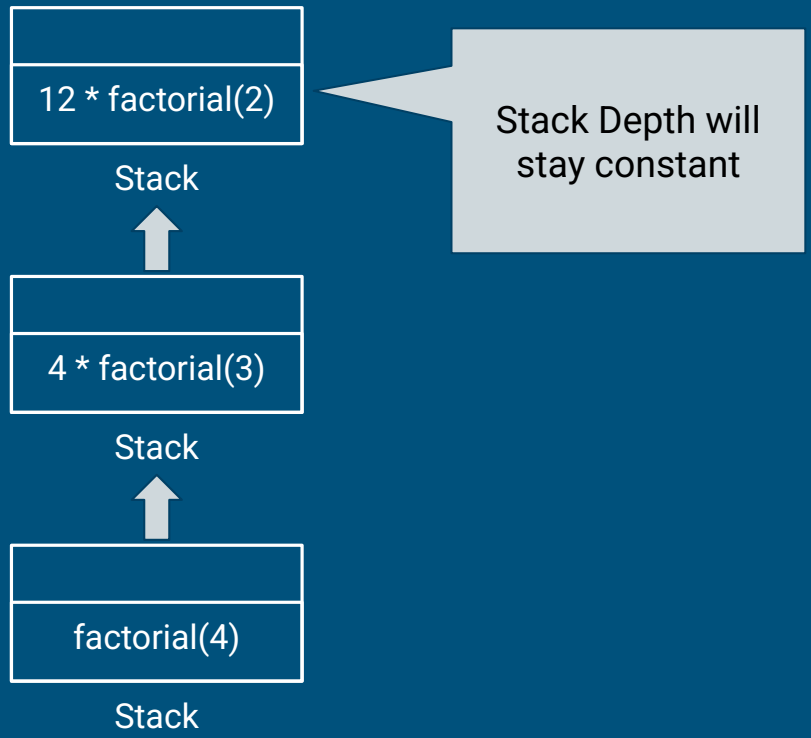
Challenge: Synthesizing Recursion



Stack Diagram

- HLS cannot determine the resource requirements for hardware implementation
- Relies on dynamic & potentially unbounded resources for its operation

Challenge: Synthesizing Recursion



Tail Call Recursion: Recursive call is the last statement in the function.

- No further computation is required after call returns

Tail Call Optimization: Transform into iterative process where the current stack frame is reused

Challenge: Synthesizing Recursion

```
int fibonacci(int n) {
    if (n == 0) {
        return 0;
    }
    ...
    return fibonacci(n - 1) +
           fibonacci(n - 2);
}
```

Cannot replace the current stack frame

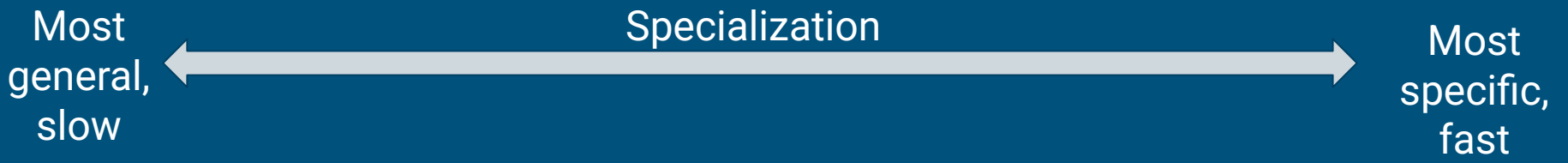
Non Tail Recursion: Operations are performed after recursive call returns

- Because computation relies on return value of recursive call, the current stack frame cannot be removed

Bambu’s existing recursion support

- Bambu’s frontend relies on GCC
- GCC can transform tail-call recursion
- Cannot transform non-tail recursion like fibonacci, ackerman, ...

Transforming Recursion to Iteration



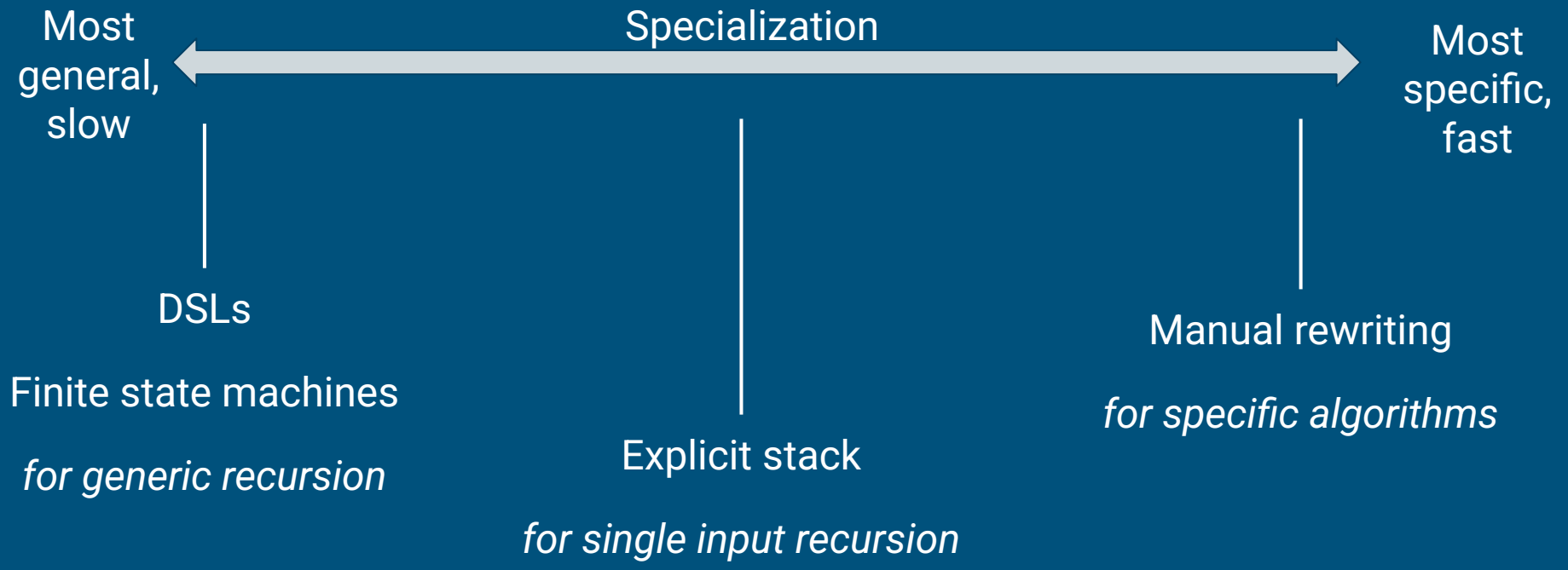
Example: Factorial

```
unsigned int factorial(unsigned int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

```
unsigned int factorial(unsigned int N) {
    int fact = 1, i;
    for (i = 1; i <= N; i++) {
        fact *= i;
    }
    return fact;
}
```

Original recursion → Manually-written iteration

Transforming Recursion to Iteration



Bambu Recurse

- Explicit stack representation to transform non-tail call, single input recursion to iteration
- Implemented inside Bambu
  - Unlike MLIR-Recursion (Li 2024)

Factorial: Designing the stack

```
unsigned int factorial(unsigned int n) {
    if (n == 1) {
        return 1;
    }
    return n * factorial(n - 1);
}
```

Explicit limit on size

```
#define MAX_STACK_SIZE 100
StackFrame stack[MAX_STACK_SIZE];
```

```
typedef struct {
    int n;
    int return_value;
} StackFrame;
```

Arguments & active variables

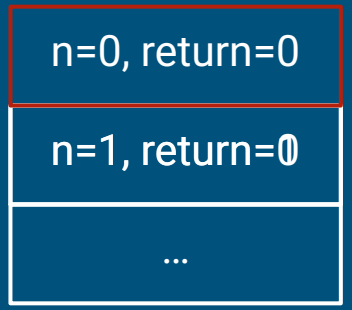
Initial values

```
push((StackFrame){.n = n,
                  .return_value = 0});
```

Factorial: Using the stack

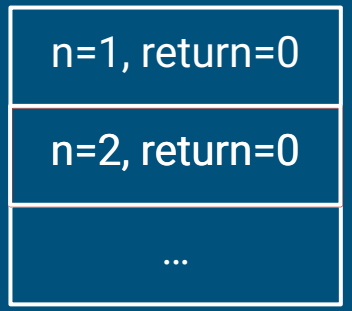
```
while (!is_empty()) {
        current_frame = pop();
        // result not ready
        if (current_frame.n == 0) {
            result = 1;
            if (!is_empty()) {
                stack[top].return_value = result;
            }
        }
```

Base case



Factorial: Using the stack

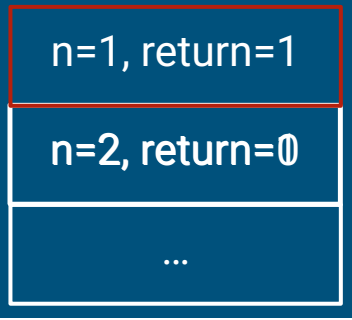
```
while (!is_empty()) {
        current_frame = pop();


        if (current_frame.n == 0) {
            ...
        }

        else {
            push((StackFrame){.n = current_frame.n, .return_value = 0});
            push((StackFrame){.n = current_frame.n - 1, .return_value = 0});
        }
```



Update recursive argument

Factorial: Using the stack

```
while (!is_empty()) {
        current_frame = pop();

        // Result is ready
        if (current_frame.return_value != 0) {

            if (!is_empty()) {
                stack[top].return_value = current_frame.n * current_frame.return_value;
            }
```



Operate on recursive result

Factorial: Using the stack

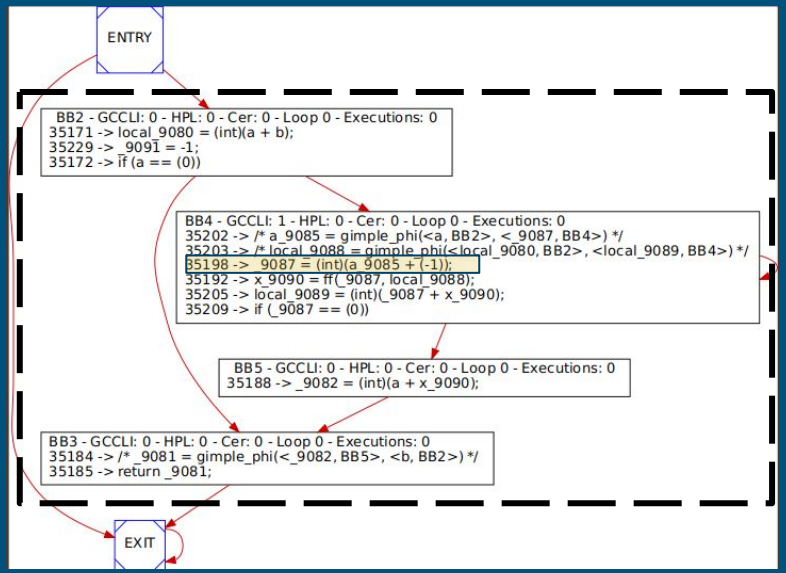
```
while (!is_empty()) {
        current_frame = pop();

        // Result is ready
        if (current_frame.return_value != 0) {

            if (!is_empty()) {
                stack[top].return_value = current_frame.n * current_frame.return_value;
            }

            else {
                result = current_frame.n * current_frame.return_value;
            }
        }
```

n=2, return=1

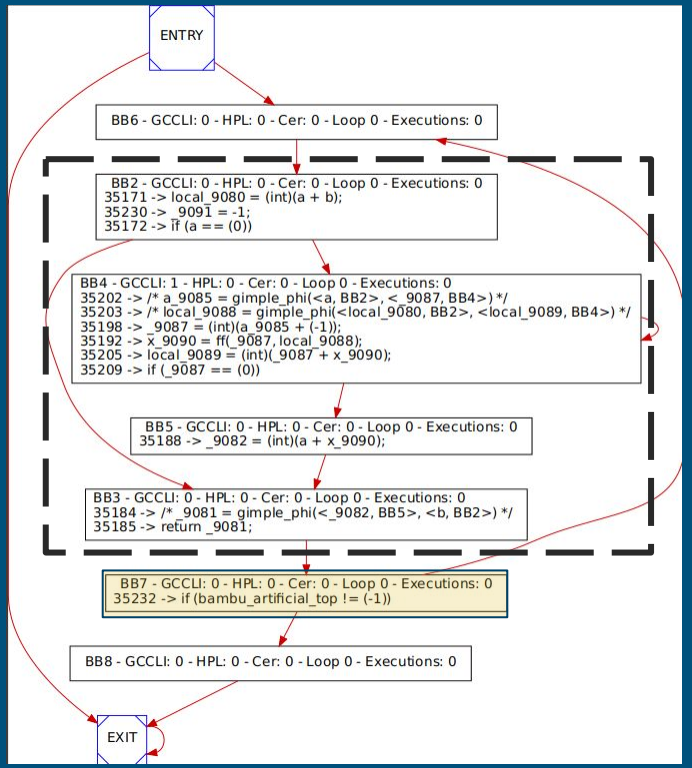
result=2

Operate on recursive result

Ongoing Work - Implementation



Original IR



Transformed IR

Ongoing Work - Implementation

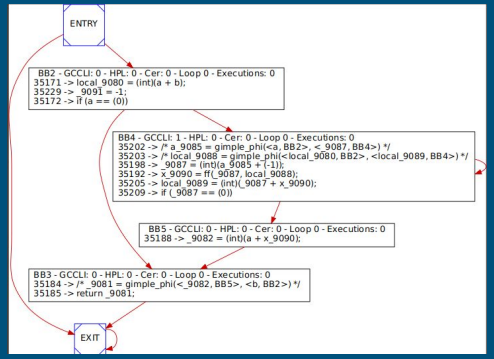
- Adding RecursionRemoval pass with explicit stack to Bambu
  - Defining StackFrame
  - Setting up stack abstraction
  - Modifying basic block IR

```
typedef struct {
    int n;
    int return_value;
} StackFrame;
```

Arguments & active variables

Ongoing Work - Challenges

- Recursion-to-iteration algorithms
- Open source software installation & documentation
- Modifying IR




```
// Insert BB_start_block into the top of IR
BB_start_block->add_pred(BB_entry->number);
BB_start_block->add_succ(first_block->number);
BB_entry->add_succ(BB_start_block->number);
first_block->add_pred(BB_start_block->number);
remove_BB(first_block->list_of_pred, 0);
remove_BB(BB_entry->list_of_succ, first_block->number);
```

Evaluation

Expressivity: What expressivity is added to the Bambu HLS?

Single-input non-tail call recursion

- Fibonacci
- Ackerman
- Heap sum
- Quicksort

Performance: Is performance comparable to manually-written iterative equivalents?

- Verilator

Supporting recursion in Bambu HLS