Automatic Recursion Elimination using Recurrence Relations for Synthesis of Stack-free Hardware

Adam Musa
McGill University
adam.musa@mail.mcgill.ca

Christophe Dubach
McGill University & Mila
christophe.dubach@mcgill.ca

***Abstract*—High-level Synthesis (HLS) eases hardware design by offering a higher level of abstraction. However, high-level programming concepts, such as recursion, are costly to synthesize, if at all possible. Recursion typically relies on a dynamic call stack, whose hardware implementation is resource-intensive and inefficient. Existing approaches solve this issue by replacing recursion with iteration using explicit stack arrays or by detecting specific patterns (*e.g.*, tail recursion) to avoid using the stack.**

This paper introduces a novel technique for transforming recursive functions into equivalent stack-free iterative implementations. Using static analysis, a recurrence relation is extracted from the function, representing it as a sequence bounded by the order of the relation. This relation is then used to optimize the process of incrementalization, constructing a synthesizable, stack-free version of the function that uses a bounded static array.

This approach is evaluated on a set of recursive benchmarks used in prior work. It eliminates recursion from 9 out of 19 benchmarks and achieves a $2.0\times$ performance speedup over state-of-the-art solutions. Additionally, it removes the need for BRAM and reduces LUT usage by 12% over prior work.

I. Introduction

Despite the wide adoption of HLS tools to program hardware, HLS still lacks support for many useful high-level programming concepts. Recursion, the backbone of many algorithms, is generally not supported by HLS. In software, recursion relies on the call stack to hold information needed to continue execution. Supporting a call stack in hardware either requires knowing the stack size ahead of time or implementing a dynamic call stack, both of which incur a large overhead.

One interesting solution consists of refactoring recursive functions into iterative implementations with statically-sized arrays. Prior work includes using dynamic analysis to limit array size [1], [2] or building efficient stack architecture [3]. Other approaches avoid the stack using tail recursion elimination or recursion unrolling [4]. These techniques share some ideas with incrementalization [5], a process more commonly used in software optimization, which constructs iterative versions of recursive functions using an extracted increment.

This paper proposes a novel method for optimizing incrementalization by eliminating usage of a stack for a larger class of recursive functions. The resulting functions only require a bounded array that can be stored in a relatively small amount of memory. This approach relies on static analysis of the abstract syntax tree (AST) to detect recursion and extract necessary information for the optimization. One key aspect of this method is the extraction of the recurrence relation from the recursive function, which is then used to find the maximum number of intermediate results that need to be stored. This number, called the order of the recursion/relation [6], is then used to create a bounded array that stores minimal information.

This paper evaluates this approach on a set of 19 benchmarks found in prior studies on supporting recursion for HLS. As we will see, it completely removes the need for a stack for 9 benchmarks. Compared to prior work [1], [2] which generate stack-based designs, the approach proposed leads to hardware that is $2\times$ faster using relatively fewer resources.

To summarize, this paper makes the following contributions:

- A more generalized incrementalization process to transform recursive programs into iterative versions.
- A technique to exploit the recurrence relation to remove the use of a stack and any resources associated with it.
- An open source tool for automatically transforming recursive C/C++ into synthesizable code, evaluated on 19 benchmarks.

II. Related Work

Despite the lack of support for recursion in modern HLS tools like Vivado, Quartus, and LegUp [12], [13], a variety of different solutions exist to tackle this problem as shown in table I. Most solutions to this problem often rely on the use and creation of a stack that is compatible with hardware.

One such solution is the creation and use of Recursive Hierarchical Finite-State Machines (RHFSM) [9], [10], [14]. These state machines are able to handle stacks and use them to execute recursive functions at the cost of extra overhead.

Another stack-based approach is found in the HLSRecurse library [3]. This library requires programs be written using its own embedded domain-specific language (DSL), which is then rewritten into synthesizable code. This produces great results, but places the effort of rewriting functions on the user.

There are also stack-free approaches to recursion, like using Runtime Reconfiguration (RTR) [11] or Recursion Flattening [8]. Both methods aim to inline recursive calls until recursion is guaranteed to complete execution. Recursion Flattening accomplishes this using a given recursion depth, while RTR reconfigures hardware dynamically during execution.

More recent approaches, like HeteroRefactor [2] and HeteroGen [1], use dynamic analysis to determine the stack size and solve other synthesis issues. They automatically refactor code and ease the use of HLS tools with general programs.

TABLE I: Overview of different approaches to handling recursion for hardware.

Approaches	This Paper	HeteroGen [1]	HeteroRefactor [2]	HLSRecurse [3]	TreeRecur [7]	Flatten [8]	RHFSM [9], [10]	RTR [11]
Automated	Yes	Yes	Yes	No	No	Yes	No	No
Prerequisites	None	None[1]	None	DSL Code	Input Size	Input Size	None	None
Method	Stack-free	Stack	Stack	Stack	Tree Stack	Unrolling	Stack	Reconfig
Application	Limited	All	All	All	Limited	Limited	All	All

[1]HeteroGen requires a Vitis HLS log file, but it can automatically extract this file if connected to Vitis HLS.

```
int fib(int n) {
    if (n <= 1) return 1;
    return fib(n - 1) + fib(n - 2);}
```

Listing 1: Recursive Fibonacci function.

```
int fib(int n) {
    int table[n] = {1,1,0,0,...};
    for(int i=2;i<n;i++)
        table[i]=table[i-1]+table[i-2];
    return table[n];
```

Listing 2: Iterative Fibonacci function with an increment of 1.

Other stack-based approaches specialize in formatting the stack for specific use cases, like TreeRecur [7], which uses a tree data structure to exploit parallelism to a greater deal.

Outside of HLS, there is a depth of research into incrementalization [15], [16] for software optimization. Moreover, tools that automate this process, like CACHET [17] and AutoInc [18] have been developed. However, these methods still rely on using a stack and stack-free solutions are constrained.

The method described in this paper reformulates the process of incrementalization into the context of HLS. Moreover, using recurrence relations, this process is improved so that stack-free solutions are available for a broader scope of programs.

III. Motivation

Consider the Fibonacci series function defined in listing 1. To synthesize this function, it needs to be transformed into an iterative function. This can be achieved naively by making the call stack explicit. A more efficient approach would be to use incrementalization/dynamic programming to find a bottom-up solution that uses tabulation [19]. A bottom-up solution uses an iterative loop that computes the result for the smallest sub-problems first, *i.e.*, the base cases, and builds up to the larger sub-problems. Meanwhile, tabulation is the process of storing intermediate results into a table, so they can be reused. The bottom-up version of the Fibonacci is shown in listing 2.

However, the table array in this function is dynamically sized as it depends on `n`. To fix this, a recurrence relation is used, which describes the current result as a function of the previous results. For the Fibonacci function the relation is:

$$fib(i) = fib(i-1) + fib(i-2) \tag{1}$$

Based on the argument of each function call this equation can be rewritten in terms of the `table` array as:

$$table[i] = table[i-1] + table[i-2] \tag{2}$$

Since earliest value used to find $table[i]$ is $table[i-2]$, the order of the relation is $i-(i-2) = 2$. This means that $table[i]$ only ever depends the previous two values, and by induction, this is true for any i. Since only 2 values need to be stored at any point, the bottom-up iterative function can be optimized:

```
int fib(int n) {
  int table[2] = {1, 1};
  for(int i = 1; i < n; i++) {
    table = {table[1], table[1]+table[0]};
  return table[1];
```

This version is now completely synthesizable into efficient hardware. To illustrate, for an input of $n = 6$, this function has a simulation runtime of 335ns, whereas the stack-based approach takes $1,475$ns — a $4.4\times$ speedup!

IV. Recursion Analysis

In order to start transforming recursive functions, there are a few steps to follow. First, the recursive functions are detected by checking for recursive calls in the AST of the program. Next, the functions are placed into one of two categories,

- Tail recursive functions, where the only recursive call is the last statement, *e.g.*, binary search.
- Non-Tail recursive functions, where the last statement is not necessarily a recursive call, *e.g.*, Fibonacci function.

Tail recursive functions are easily transformed using a tail call optimization, commonly found in some HLS tools [20]–[22], which replaces function execution with the recursive call using goto statements or a while loop. The non-tail recursive functions are then further divided into recurrence sequences (section V) and divide-and-conquer functions (section VI). Before processing the functions any further, a copy-propagation pass and constant folding are used to simplify the functions to simplify later analysis and transformations.

V. Recurrence Sequences

Recurrence sequences are functions defined in terms of a sequence. Recurrence sequences in code are of the form:

```
int func(a1, a2, ..., am) {
    if (...) {...} ... // Base Cases
    else if (...) { ... // Recursive Body
        int r1 = f(h_11(a_1),...,h_1m(a_m));
        ... // Recursive Calls
        int rk = f(h_k1(a_1),...,h_km(a_m));
        ...
        return g(r1,...,rm);}
    ...} // Other Code (Possibly Recursive)
```

Each function h_t defines the arguments of each recursive call and, as shown later, they must be invertible. Section V-A covers single-parameter functions, and section V-B generalizes the process to multi-parameter functions.

A. Functions with One Parameter

These functions are of the form:

```
int f(a) {
    if (...) {...} ...  // Base Cases
    else if (...) { ...
        int r1 = f(h_1(a));
        ... // Recursive Calls
        int rm = f(h_m(a));
        ...
        return g(r1,...,rm);}
```

The first step is using simple incrementalization. This automated transformation produces a dynamic array to store intermediate results and a loop with an increment of 1 to find every value needed. This results in the following function:

```
int f(a) {
  int table[a];
  table[0],...,table[b] = ...; // Store Base Cases
  for (int i=b; i <= a; i++) {
    if (...) {...} ...
    else {
        int r1 = table[h_1(i)];
        ...
        int rm = table[h_m(i)];
        ... // Add new result to table
        table[i] = g(r1,...,rm);}
  return table[a];
```

To synthesize this function, the `table` array needs to be statically bound. The first step to achieving this is extracting the recurrence relation of the function in terms of the array `table`. The `table` form of this relation is:

$$table[i] = g(table[h_1(i)], ..., table[h_m(i)]) \tag{3}$$

If the functions $h_t(i)$ only subtract i by a constant then:

$$table[i] = g(table[i - k_1], ..., table[i - k_m]) \tag{4}$$

In this case, the array is simply bounded by k_{max} (the largest k_t), since only the last k_{max} values need to be remembered.

However, when other operations are involved, this bound can be difficult to find. To solve this problem, the values stored in the array need to be changed such that each $h_t(i)$ is reformulated into the form $i - k_t$. This can be accomplished by changing the loop update (`i++`). To illustrate, if $table[i] = g(table[\frac{i}{2}], table[\frac{i}{4}])$, the only values used are $\{a, \frac{a}{2^1}, \frac{a}{2^2}, ...\}$. If the loop update is changed to $i * 2$ then, instead of storing every value $\{b, ..., a\}$, the values stored in the array are $\{b, b * 2, b * 2^2, ...\}$ (assuming b is the base case). So to access $table[\frac{i}{2}]$ under this context, $table[i - 1]$ can be used. This means that the relation is now $table[i] = g(table[i - 1], table[i - 2])$ and the array can be bounded.

To generalize this, first assume there are only two recursive calls with two functions $h_1(i)$ and $h_2(i)$. One way of reformulating these functions is finding a function $S(i)$ that can define both. To explain, this function S should first satisfy:

$$h_1(i) = S^{(k_1)}(i) \tag{5}$$

$$h_2(i) = S^{(k_2)}(i) \tag{6}$$

Here $S^{(k_t)}$ denotes applying S to i repeatedly k_t times. Given S, the next step is defining the update for i. Since both functions h_1 and h_2 only use compositions of S, only values found by repeatedly applying S need to be stored, *i.e.*, $\{..., S^2(i), S(i), i\}$. This array stores all the values needed for the recursion; however, it is defined backwards relative to i (the latest value). To go forward and update i, the opposite needs to be done. As such, the update for i should be the inverse of S, which also means the functions h_t must be invertible.

Now, the array `table` only contains values defined by S^{-1}. As such, with $h_1(i) = S^{(k_1)}(i)$, the entry $table[h_1(i)]$ is now $-k_1$ positions from i and $table[h_1(i)]$ should be substituted with $table[i - k_1]$. The same can be done with $table[h_2(i)]$ and k_2, so the array can again be bounded by k_{max}.

All this leads to the critical step: finding this function $S(i)$. As mentioned previously, this function S must define the both functions h_1 and h_2. In other words, h_1 and h_2 can be constructed using repeated applications of S as shown in eq. (5) and eq. (6). For example, if $h_1(i) = \frac{i/2-1}{2} - 1$ and $h_2(i) = \frac{i}{2} - 1$, then $S(i) = \frac{i}{2} - 1$ results in $h_1(i) = S^{(2)}(i)$ and $h_2 = S^{(1)}(i)$. Using this information, there are a variety of methods (solvers, AST analysis, etc.) of finding S.

This process is then generalized to handle functions with more than two recursive calls. In general, for all the functions $h_1, ..., h_m$ across each recursive call, this must be true:

$$h_t(i) = S^{(k_t)}(i) \qquad t \in \{1, ...m\} \tag{7}$$

The function S only exists if it is able to define each function $h_1, ..., h_m$. The values of k_t can then be found for each function h_t and the inverse of S can be used as the update for i. The bound k_{max} is found and applying these transformations to the original incrementalized function produces:

```
int f(a) {
  int table[k_max] = ...; // Store Base Cases
  table[0],...,table[b] = ...; // Store Base Cases
  for (int i=b; i <= a; S_inv(i)) {
    if(...) { ...
      int r1 = table[-k_1];
      ... // Recursive calls
      int rm = table[-k_m];
      ...
      shift(table); table[k_max-1] = res;}}
  return table[k_max-1]}
```

This function is now synthesizable as it has no recursion and a statically sized array. Note, for recursive calls involving division, `S_inv(i)` may not be enough to recover the true value of i due to truncation/rounding. In such cases, the function S can be looped to find i. For example, if the original parameter is $a = 15$, $i = 1$ and $S(i) = \frac{i}{2}$. Then the update $i = S^{(2)}(15) = \frac{15}{4} = 3$ is used to avoid the truncation error.

B. Functions with Multiple Parameters

These functions are of the form:

```
int f(a1,...,az) {
    if (...) {...} .... // Base Cases
    else if (...) { ...
        int r1 = f(h_11(a1), ..., h_z1(az));
        ...
        int rm = f(h_1m(a1), ..., h_zm(az));
        ...
        return res;}}
```

The bottom-up incrementalized version looks like this:

```
int f(a1,...,az) {
  int table[a1, ..., az] = ...;
  for (int i1=b1; i < a1; i1++) {
    ... // For loops for each parameter
      for (int iz=bz; i < az; iz++) {
        if (...) {...}
        else if (...) {...
            int r1 = table[h_11(i1),...,h_z1(iz)];
            ... // Access previous results
            int rm = table[h_1m(i1) ...,h_zm(iz)];
            ...
            table[i] = res;}}}
  return table[a1]...[az]
```

This function still uses a dynamic array/stack to hold intermediate values. This version of the problem is solved by dividing the process into multiple single parameter cases and the method in the previous section is used on each case. To accomplish this, the functions h_{tp} are separated based on their corresponding parameter (based on each dimension in the array), producing the lists $\{h_{11},...,h_{1m}\}$,..., $\{h_{z1},...,h_{zm}\}$. The process in section V-A is then repeated on each list of functions $\{h_{p1},...,h_{pm}\}$. Each list will have its own increment, defined by an inverse function $S^{-1}_{h_p}$, and a position k_{pt} for the functions $\{h_{p1},...,h_{pm}\}$.

Next, each array access $table[h_{1t}(i1),...,h_{zt}(iz)]$ must be reformulated to be of the form $table'[i'-k_t]$, where $table'$ is a one dimensional array. This means each array access needs a unified position k_t. The functions are grouped based on array access forming the sets $\{h_{11},...,h_{z1}\}$,..., $\{h_{1m},...,h_{zm}\}$. Each function in the list $\{h_{1t},...,h_{zt}\}$ has its own position $k_{1t},...,k_{zt}$. The position of the array access k_t only exists if all the positions are equal, then $k_t = k_{1t} = ... = k_{zt}$.

Next, the incremental loop is constructed. A counter tracks the iterations and the intermediate value of each parameter is found using its corresponding increment $(S^{-1}_{h_1},...,S^{-1}_{h_z})$. The starting values of each parameter are found using the number of iterations and the non-inverted functions $S_{h_1},...,S_{h_z}$. The total number of iterations is the minimum number of times $S_{h_1},...,S_{h_z}$ can be applied on the original parameters until a base case is reached. Combining all this, the new function is:

```
int f(a1,...,az) {
  int table[kp] = ... ; // Store Base Cases
  int num_iter_a1= find_iter(a1, conds_1, S1_inv);
  ... // Find number of iterations
  int num_iter = min(num_iter_a1,..., num_iter_az)
  int a1_val = repeat(S1, input=a1, num=num_iter)
  ... // Find all initial values
  for (int i=0; i < num_iter; i++) {
    if (...) {...
    else if (...) {...
        int r1 = table[k_1];
        ...
        shift(table); table[kp] = res;}
    a1_val = S1_inv(a1_val);
    ...} // Update all the parameters
  return table[kp];}
```

With the previous section, this covers a wide scope of recursive functions and produces iterative, hardware synthesizable code.

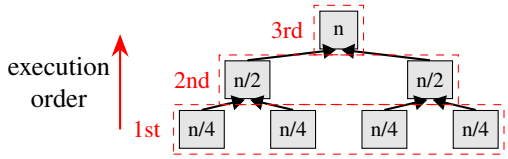

Fig. 1: Bottom-up execution order.

VI. Divide-and-Conquer

The divide-and-conquer functions supported are of the form:

```
void divconq(data, int start, int end) {
    if (end-start <= n0) {...} // Base Cases
    else { ... // No changes to data
        divconq(data, start, start+k)
        ... // Other Recursive Calls
        divconq(data, start+n*k, end)
        ...}} // Remaining Statements
```

Here, $data$ is compound data (array, string, etc.), and $start$ and end are the indices of its beginning and end. In this code, $n0$ is the maximum size for base-case subproblems. The transformation for these functions is based on a bottom-up approach that executes every smaller problem before increasing the problem size as shown in fig. 1. This leads to the following constraints:

1) The function must have no return value.
2) The recursion is not generative.
3) There must be one compound data parameter and two indices pointing to the start and end of this data.
4) Divided into disjoint sub-problems of the same size.

The first constraint ensures no dynamic memory is needed (not HLS supported), otherwise all the intermediate results (up to $log(end-start)$) need to be stored during the bottom-up execution. The second constraint states that the recursion cannot be generative. Generative recursion is recursion where the behavior of the function changes based on run-time information, *e.g.*, the pivot in Quicksort. The recursive calls in these functions cannot be statically analyzed making it incompatible with this approach. The last two constraints simplify the analysis and transformation. Given these constraints hold, the function can be transformed.

In the function, the original problem size is $n = end-start$. The first function call produces k sub-problems of size $\frac{n}{k}$, and each sub-problem produces k calls of size $\frac{n}{k^2}$. This continues and if the function has a recursion depth of $d = \lfloor log_k(n) \rfloor - \lfloor log_k(n0) \rfloor$, there are k^d base cases. The recursion depth is the number of recursive calls until the base case is reached.

The first step is to execute all the k^d base cases of size $n0$. This means executing every non-overlapping combination of $start$ and end of size $n0$. This is accomplished by extracting the function body, removing the recursive calls, and wrapping it in a loop that iterates through the base cases.

After completing the loop, the size of the sub-problem can be increased to $n_0 \cdot k$. The same steps are repeated, except the loop now iterates through problems of size $n_0 \cdot k$. This process is repeated for each problem size until the final loop is iterates over the original size of $n0 \cdot k^d = n$. In all the cases, removing the recursive calls is possible since every recursive call/sub-problem is already executed due to the bottom-up approach.

TABLE II: Resource Usage across all approaches and benchmarks

Type	Benchmark	HeteroRefactor [2] LUT	FF	BRAM	Freq	HLSRecurse [3] LUT	FF	BRAM	Freq	This Approach LUT	FF	BRAM	Freq
Tail	Binary Search	**NS:** Pointer-to-Pointer Error				Synthesis Error				604	304	0Kb	189
	GCD	2,963	2,744	162Kb	333	2,222	2,518	54Kb	386	**2,009**	**2,385**	**0Kb**	**531**
Recur-Seq	Factorial	1,057	196	126Kb	154	448	160	18Kb	**229**	**268**	**136**	**0Kb**	215
	Fibonacci	1,218	231	162Kb	268	725	214	72Kb	410	**273**	**168**	**0Kb**	**530**
	Modified Fibonacci	1,790	336	198Kb	184	**974**	**245**	90Kb	319	1,142	973	**0Kb**	**529**
	List Sum	**NS:** Pointer-to-Pointer Error				Synthesis Error				325	174	0Kb	382
	Ackermann	1,548	**150**	234Kb	263	**610**	176	**54Kb**	**410**	**NP:** Non-primitive Recursion			
Divide-and-Conquer	MS	**NS:** Pointer-to-Pointer Error				Synthesis Error				2881	2,008	36Kb	142
	MS-Ternary	**NS:** Pointer-to-Pointer Error				Synthesis Error				5203	2985	48Kb	143
	QuickSort[1]	**NS:** Pointer-to-Pointer Error				1,216	439	90Kb	258	**NP:** Generative Recursion			
	HeapSum (HS)[1]	**NS:** Pointer-to-Pointer Error				**976**	**147**	72Kb	137	989	763	**0Kb**	**202**
	TiledMM[1]	Transformation Error				2,793	818	108Kb	118	**NP:** Does not Follow Constraints			
	FFT[1]	Transformation Error				3,507	2,832	180Kb	117	**NP:** Does not Follow Constraints			
	MISER[1]	Transformation Error				11,082	5,298	216Kb	109	**NP:** Generative Recursion			
	Strassen	16,351	7,257	1,944Kb	137	**NP:** Uses Dynamic Memory				**NP:** Generative Recursion			
Back-tracking	AC	11,827	4,502	1,944Kb	146	**NP:** Uses Dynamic Memory				**NP:** Algo cannot be Bottom Up			
	DFS	3,143	903	360Kb	146	**NP:** Uses Dynamic Memory				**NP:** Algo cannot be Bottom Up			
	Sudoku	**NS:** Pointer-to-Pointer Error				Synthesis Error				**NP:** Generative Recursion			
	LinkedList	6,959	2,775	756Kb	146	**NP:** Uses Dynamic Memory				**NP:** Algo cannot be Bottom Up			

*Frequency (Freq) is in MegaHertz (MHz). *NS = Not Supported *NP = Not Possible using this method
[1]Synthesized in Vivado HLS 2018.1

Combining all these steps results in the following loops:

```
for (int i = init_start; i < n; i = i + n0) {
    start = i; end = min(i + n0, n);
    if (end-start <= n0) {...} // Base Cases
    else { ...}} // No Recursive Calls
... // For loop for every size
for (int i = init_start; i<n; i = i + n0 * k**d) {
    start = i; end = min(i + n0*k**d, n);
    if (end-start <= n0) {...} // Base Cases
    else {...}} // No Recursive Calls
```

These loops have the same structure, with the only change being the problem size $n0 * k^p$ going from $p = 0$ to $p = k^d$. As such, a loop fusion optimization is used to produce:

```
void divconq(x, ..., start, end) {
   init_start = start; n = end-start;
   recursion_depth = log(n, k);
   for (int p = 0; p < recursion_depth; p += 1) {
      for (int i=init_start; i<n; i=i+n0*(k**p)) {
         start = i;
         end = min(i + n0 * k**p, n);
         if (end - start <= 1 && ...) {...}
         else {...}// With No Recursive calls
```

The implementation includes a few extra lines for variable management that are omitted for brevity. This function is stack and recursion free, leading to synthesizable hardware.

VII. Experimental Setup

A. Benchmarks

The approach in this paper is compared against two state-of-the-art prior works, HeteroRefactor [2] and HLSRecurse [3]. They are tested on a collection of 19 benchmarks, some of which were used in prior work [1]–[3]. Of these, Modified Fibonacci and Ternary Mergesort (MS) are synthetic and used to show generalizability. This method correctly synthesizes 9 benchmarks and all the results are reported in table II. Benchmarks marked as **NP** (not possible) are not synthesizable using the specific tool's methodology, either due to synthesis issues (*e.g.*, dynamic memory) or method restrictions. Benchmarks marked as **NS** (not supported) should be synthesizable, but a bug or lacking implementation causes failure.

B. Environment and Implementation Details

This approach is implemented in C++ and uses ROSE [23] to translate programs into ASTs. The Python solving library SymPy [24] is used as a solver. The resulting code is synthesized and simulated using Vitis HLS 2022.2, or using Vivado HLS 2018.1 in the case of bugs for the other approaches. The designs are targeted to a Xilinx Virtex Ultrascale+ XCVU11P FPGA with a frequency of 1000 MHz. The LUT, FF, and BRAM (in Kilobits) usage is reported, while DSP usage is omitted as it is constant across approaches. The run-times of this approach and HeteroRefactor are also omitted as they only take a few seconds. The implementation of this work covers a limited, simpler version of the method presented in this paper, where only subtraction and division are considered as valid updates for the arguments of a recursive call. The code for the implementation and benchmarks is available as open-source[1].

VIII. Evaluation

A. Coverage

As seen in table II, the implementation of this approach successfully transforms 9 out of the 19 benchmarks. The Ackermann function contains nested recursive calls and is a non-primitive recursive function (cannot be computed using "for" loops). Tiled Matrix Multiplication and Fast Fourier Transform both do not fall within the divide-and-conquer constraints, but future work and further research can look into making incrementalization possible. Depth-first search and linked list

[1]https://github.com/AdamMoMu/RecRel_RecElim

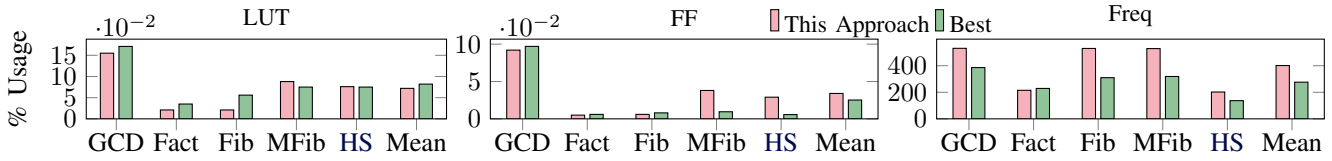

Fig. 2: Percent resource usage for this approach and the best result between HLSRecurse and HeteroRefactor

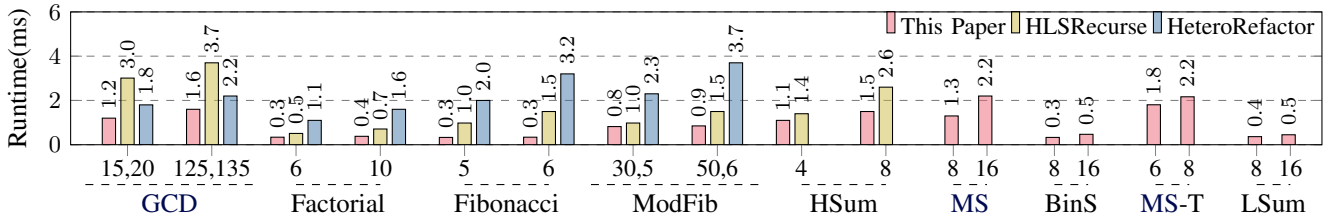

Fig. 3: Simulation runtime on varying inputs sizes of benchmarks synthesizable by this approach.

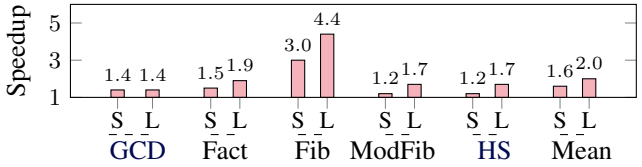

Fig. 4: Speedup over the best results for small and large inputs.

cannot be implemented bottom-up as it completely changes the algorithm (Aho-corasick's helper functions have the same issue). The other functions are generative and cannot be incrementalized. In practice, HeteroRefactor (or another tool) can be used as fallback for any unsuccessful transformation.

Of the other methods, HeteroRefactor had some transformation and synthesis errors, mostly due to adding pointers to pointers into programs which are non-synthesizable constructs. HLSRecurse programs resulted in correct code when executed in software, but faced synthesis errors for some functions that operate on arrays. Some benchmarks are also exclusively synthesizable by HeteroRefactor, due to its ability to handle other synthesis issues (*e.g.*, dynamic memory).

B. Resource Usage and Clock Frequency

We turn our attention to the results of the synthesis in table II. The designs generated using this method use 0Kb of BRAM, with the exception of Mergesort allocating BRAM to a helper function. Consequently, this approach avoids any out-of-memory errors related to recursion.

Figure 2 shows the percent usage of the resources, comparing this work to the best-performing design among HeteroRefactor and HLSRecurse. BRAM usage is omitted as it is constant for this paper and input dependent for the other approaches. In this approach, the stack is substituted by a small array, so FF (Flip-Flop) usage increases by 35% as reflected in fig. 2. However, this is a side-effect of recursive calls with a division operation, and in the other cases, there is actually a decrease in FF usage. Moreover, due to the absence of a stack, we also see a 12% decrease in LUT usage and 45% increase in clock frequency. Overall, the method in this paper accomplishes its goal of removing BRAM usage, while not heavily impacting other resource usage.

C. Performance

The simulation runtime is shown in fig. 3 and the speedup over the fastest result is reported in fig. 4. This approach outperforms the other approaches on all their common benchmarks, showing an average speedup of $1.6\times$ for small inputs and $2.0\times$ for large inputs. Moreover, due to the bottom-up approach, as the input gets larger, the speedup gets larger, as other approaches require exponentially more recursive calls.

IX. Limitations and Future Work

This paper introduces a novel method for optimizing incrementalization for HLS. However, this method is limited in scope and there are still optimizations to be made (large FF usage). Specifically, for divide-and-conquer functions (section VI), only the first two constraints mentioned are necessary. Further research into and improvements to incrementalization could help broaden the scope of this work. Moreover, combining ideas from this paper with other work, like the dynamic analysis in HeteroRefactor, could also prove beneficial. In addition, the methods in this paper could be adapted and applied to non-HLS fields. Finally, while this paper gathered a set of benchmarks, HLS recursion benchmarks are still lacking. Further work is needed to develop more standardized benchmarks that cover a greater scope of recursive functions.

X. Conclusion

Recursion is a popular method for algorithm design, as such increasing support for recursion in HLS makes hardware design easier to adopt. This paper has shown that transforming recursive functions into stack-free iterative versions is not limited to tail recursion. It has proposed new methods that tackle a wider scope of recursive functions and produce efficient, synthesizable, stack-free code. The approach has been tested on a common set of benchmarks, reaching a speedup of $2.0\times$ when applicable, and substantially better scalability.

Acknowledgement

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grants Program [grant RGPIN-2020-05889], and the Canada CIFAR AI Chairs Program.

References

[1] Q. Zhang, J. Wang, G. H. Xu, and M. Kim, "Heterogen: transpiling c to heterogeneous hls code with automated test generation and program repair," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. Association for Computing Machinery, 2022, p. 1017–1029. [Online]. Available: https://doi.org/10.1145/3503222.3507748

[2] J. Lau, A. Sivaraman, Q. Zhang, M. A. Gulzar, J. Cong, and M. Kim, "Heterorefactor: refactoring for heterogeneous computing with fpga," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. Association for Computing Machinery, 2020, p. 493–505. [Online]. Available: https://doi.org/10.1145/3377811.3380340

[3] D. B. Thomas, "Synthesisable recursion for c++ hls tools," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 91–98.

[4] I. Skliarova and V. Sklyarov, "Recursion in reconfigurable computing: A survey of implementation approaches," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 224–229.

[5] Y. A. Liu and S. D. Stoller, *Dynamic Programming via Static Incrementalization*. Springer Netherlands, 2008, pp. 71–92. [Online]. Available: https://doi.org/10.1007/978-1-4020-6585-9_9

[6] A. Doerr and K. Levasseur, *Applied Discrete Structures*, 3rd ed. Alan Doerr & Kenneth Levasseur, 2024. [Online]. Available: https://discretemath.org/ads/index-ads.html

[7] B. Morrison and M. Łukowiak, "Tree-based hardware recursion for divide-and-conquer algorithms," in *2021 28th International Conference on Mixed Design of Integrated Circuits and System*, 2021, pp. 147–152.

[8] G. Stitt and J. Villarreal, "Recursion flattening," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, ser. GLSVLSI '08. Association for Computing Machinery, 2008, p. 131–134. [Online]. Available: https://doi.org/10.1145/1366110.1366143

[9] D. Mihhailov, V. Sklyarov, I. Skliarova, and A. Sudnitson, "Parallel fpga-based implementation of recursive sorting algorithms," in *2010 International Conference on Reconfigurable Computing and FPGAs*, 2010, pp. 121–126.

[10] V. Sklyarov, "Fpga-based implementation of recursive algorithms," *Microprocessors and Microsystems*, vol. 28, no. 5, pp. 197–211, 2004, special Issue on FPGAs: Applications and Designs. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0141933104000201

[11] G. Ferizis and H. E. Gindy, "Mapping recursive functions to reconfigurable hardware," in *2006 International Conference on Field Programmable Logic and Applications*, 2006, pp. 1–6.

[12] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. Association for Computing Machinery, 2011, p. 33–36. [Online]. Available: https://doi.org/10.1145/1950413.1950423

[13] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, Sep. 2013. [Online]. Available: https://doi.org/10.1145/2514740

[14] V. Skylarov, I. Skilarova, and B. Pimentel, "Fpga-based implementation and comparison of recursive and iterative algorithms," in *International Conference on Field Programmable Logic and Applications, 2005.*, 2005, pp. 235–240.

[15] Y. A. Liu and S. D. Stoller, "From recursion to iteration: what are the optimizations?" *SIGPLAN Not.*, vol. 34, no. 11, p. 73–82, Nov. 1999. [Online]. Available: https://doi.org/10.1145/328691.328700

[16] Y. A. Liu and T. Teitelbaum, "Caching intermediate results for program improvement," in *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ser. PEPM '95. Association for Computing Machinery, 1995, p. 190–201. [Online]. Available: https://doi.org/10.1145/215465.215590

[17] Y. Liu, " CACHET: an interactive, incremental-attribution-based program transformation system for deriving incremental programs ," in *Knowledge-Based Software Engineering Conference*. IEEE Computer Society, Nov. 1995, p. 19. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/KBSE.1995.490115

[18] Y. Zhang and Y. A. Lin, "Automating derivation of incremental programs," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98. Association for Computing Machinery, 1998, p. 350. [Online]. Available: https://doi.org/10.1145/289423.289480

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. MIT Press, 2022.

[20] P. Ferreira, J. C. Ferreira, and J. C. Alves, "Erlang inspired hardware," in *2010 International Conference on Field Programmable Logic and Applications*, 2010, pp. 244–246.

[21] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in haskell," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98. Association for Computing Machinery, 1998, p. 174–184. [Online]. Available: https://doi.org/10.1145/289423.289440

[22] E. D. Sozzo, D. Conficconi, A. Zeni, M. Salaris, D. Sciuto, and M. D. Santambrogio, "Pushing the level of abstraction of digital system design: A survey on how to program fpgas," *ACM Comput. Surv.*, vol. 55, no. 5, Dec. 2022. [Online]. Available: https://doi.org/10.1145/3532989

[23] D. Quinlan and C. Liao, "The ROSE source-to-source compiler infrastructure," in *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, vol. 2011. Citeseer, 2011, p. 1.

[24] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

[25] E. A. Boiten, "Improving recursive functions by inverting the order of evaluation," *Sci. Comput. Program.*, vol. 18, no. 2, p. 139–179, Apr. 1992. [Online]. Available: https://doi.org/10.1016/0167-6423(92)90008-Y

[26] J. Cong, J. Lau, G. Liu, S. Neuendorffer, P. Pan, K. Vissers, and Z. Zhang, "Fpga hls today: Successes, challenges, and opportunities," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 15, no. 4, Aug. 2022. [Online]. Available: https://doi.org/10.1145/3530775

[27] J. Arsac and Y. Kodratoff, "Some techniques for recursion removal from recursive functions," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 2, p. 295–322, Apr. 1982. [Online]. Available: https://doi.org/10.1145/357162.357171

[28] W. H. Press and G. R. Farrar, "Recursive stratified sampling for multidimensional monte carlo integration," *Computer in Physics*, vol. 4, no. 2, pp. 190–195, 03 1990. [Online]. Available: https://doi.org/10.1063/1.4822899

[29] R. S. Bird, "Tabulation techniques for recursive programs," *ACM Comput. Surv.*, vol. 12, no. 4, p. 403–417, Dec. 1980. [Online]. Available: https://doi.org/10.1145/356827.356831

[30] L. Gauthier, T. Ishihara, and H. Takada, "Stack frames placement in scratch-pad memory for energy reduction of multi-task applications," 01 2010.

[31] Y. A. Liu, *Principled strength reduction*. Springer US, 1997, pp. 357–381. [Online]. Available: https://doi.org/10.1007/978-0-387-35264-0_14

[32] Y. A. Liu and S. D. Stoller, "Optimizing ackermann's function by incrementalization," *SIGPLAN Not.*, vol. 38, no. 10, p. 85–91, Jun. 2003. [Online]. Available: https://doi.org/10.1145/966049.777398

[33] Y. A. Liu, "Incremental computation: What is the essence? (invited contribution)," in *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation*, ser. PEPM '24. ACM, Jan. 2024, p. 39–52. [Online]. Available: http://dx.doi.org/10.1145/3635800.3637447