10-2020

Tree-Based Hardware Recursion for Divide-and-Conquer Algorithms

Braeden Morrison
bjm4184@rit.edu

Tree-Based Hardware Recursion for Divide-and-Conquer Algorithms

Braeden Morrison

Tree-Based Hardware Recursion for Divide-and-Conquer Algorithms

Braeden Morrison

October 2020

A Thesis Submitted
in Partial Fulfillment
of the Requirements for the Degree of
Master of Science
in
Computer Engineering

RIT | Kate Gleason College of Engineering

Department of Computer Engineering

Tree-Based Hardware Recursion for Divide-and-Conquer Algorithms

Braeden Morrison

Committee Approval:

Marcin Łukowiak *Advisor* Date
Department of Computer Engineering

Matthew Fluet Date
Department of Computer Science

Sonia López Alarcón Date
Department of Computer Engineering

Acknowledgments

I would like to thank all of my friends and family for supporting and encouraging me throughout my entire college career.

I would also like to thank all of my peers in the High Performance Computing and Applied Cryptography and Information Security labs for giving me a place to discuss my research and receive valuable insights and advice.

Finally, I would like to that all of the members of my committee, as without their advice and feedback this thesis would not have been possible.

Abstract

It is well-known that custom hardware accelerators implemented as application-specific integrated circuits (ASICs) or on field-programmable gate arrays (FPGAs) can solve many problems much faster than software running on a central processing unit (CPU). This is because FPGAs and ASICs can have handcrafted data and control paths which exploit parallelism in ways that CPUs cannot. However, designing custom hardware is complicated and implementing algorithms in a way that takes advantage of the desired parallelism can be difficult. One class of algorithms that exemplifies this is divide-and-conquer algorithms.

A divide-and-conquer algorithm is a type of recursive algorithm that solves a problem by repeatedly dividing it into smaller sub-problems. These algorithms have a lot of parallelism because they generate a large number of sub-problems that can be computed independently of each other. Unfortunately, traditional stack-based approaches to handling recursion in hardware make exploiting this parallelism difficult.

This work proposes a new general-purpose approach to implementing recursive functions in hardware, which we call TreeRecur. TreeRecur uses trees to represent the branching recursive function calls of divide-and-conquer algorithms, which makes it possible to take advantage of their procedure-level parallelism. To allow for design flexibility, TreeRecur executes algorithms using a configurable number of independent function processors. These processors are generated using high-level synthesis, making it easy to implement a variety of different algorithms.

Our solution was tested on three different algorithms and compared against software implementations of the same algorithms. Performance results were collected in terms of execution speed and energy consumption. TreeRecur was found to have execution speeds comparable to software when differences in clock speed were accounted for and was found to consume up to 11.2 times less energy.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Hardware implementations of many computationally intensive algorithms tend to be much faster and more efficient than their software equivalents. This is a result of the greater levels of flexibility and parallelism that are possible in hardware compared to software running on the fixed architecture of a central processing unit (CPU). However, the greater flexibility of hardware design means that careful consideration must be put into implementing abstractions that are taken for granted in software. One abstraction in particular that has caused trouble for the hardware design community is recursion.

Many different hardware implementations for recursion have been proposed over the years [5] and each has its own strengths and weaknesses. The most popular solutions are the stack-based ones, which are space-efficient and general enough to implement any algorithm [2, 3]. The main problem with these approaches is that they struggle to exploit procedure-level parallelism.

This is an issue for a subset recursive algorithms, called divide-and-conquer algorithms, which solve a problem by recursively dividing it into smaller sub-problems. Divide-and-conquer algorithms generate a large number of independent procedure calls as they execute and existing stack-based implementations of recursion cannot

compute them in parallel. There are alternative implementations of recursion that can exploit the procedure-level parallelism of divide-and-conquer algorithms [6, 7], but they consume far more area than stack-based solutions and are less flexible in the algorithms they support.

1.2 Objective

The goal of this research is to propose a new solution for implementing recursive algorithms in hardware, which we call TreeRecur, that uses a flexible tree-based approach to recursion. The tree-based implementation can exploit the procedure-level parallelism of divide-and-conquer algorithms, while keeping most of the flexibility and small design sizes that make the stack-based approaches popular.

Chapter 2

Background

2.1 Tree Data Structures

Trees are a common type of data structure that are often used to establish an hierarchy amongst their elements [8]. A tree is a collection of elements, called *nodes*, where each node stores some data and a set of trees, which are called the node's *children*. The node at the top of the hierarchy is known as the *root*. In Figure 2.1, the root node is the node labeled `7`, and the nodes labeled `3`, `10`, and `4` are its children.

In a tree, the node directly above another node in the tree is called that node's *parent*. The root of a tree is special because it is the only node in a tree that has no parent. Conversely, nodes without any children are called *leaf* nodes and represent the end of a branch of the tree. In Figure 2.1 nodes `8`, `10`, `12`, and `2` are all leaf nodes.

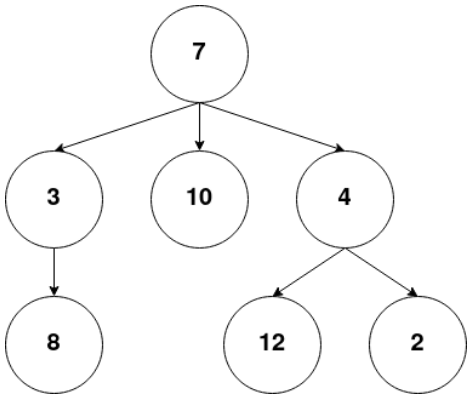


Figure 2.1: A basic tree structure with its root at `7` and leaves at `8`, `10`, `12`, and `2`.

The nodes of the tree in Figure 2.1 have different numbers of children. The number of children that a node has is called the *degree* of the node. In general, each node in a tree can have its own degree, but it is often useful to construct trees where all of the nodes have the same degree, which are called *N-ary trees*. A tree where all of the nodes have a degree of 2 is called a *binary tree*, when all of the nodes have a degree of 3 it is called a *ternary tree*, and so on.

2.1.1 Tree Implementations

Trees are an abstract data structure and have no fixed implementation, but a common approach can be seen in Algorithm 2.1, which shows a binary tree data structure for a C-like programming language. The *left* and *right* elements are pointers to the node's

Algorithm 2.1 Binary Tree Data Structure with Integers

```
1: struct Tree {
2:     int     data;
3:     Tree*   left;
4:     Tree*   right;
5: };
```

two children while the *data* element stores the data associated with the node. For a leaf node, both the *left* and *right* pointers would be set to `NULL` to represent that the node has no children.

As a tree is used, there is often a need for the tree grow or shrink as more or fewer elements need to be stored. With a linked tree implementation such as the one shown in Algorithm 2.1, adding and removing nodes is fairly straightforward. An existing node can be attached to a tree by locating a node where one or both of its child pointers are `NULL` and setting that pointer to be the address of the new node. Similarly, a leaf node can be removed from the tree by setting its parent's child pointer to `NULL`.

2.1.2 Array-Based Tree Implementation

With a linked tree implementation, adding existing nodes to the tree is easy, but generating new tree-nodes can sometimes be problematic. In software programs running on operating systems, memory for new tree nodes can be allocated and freed using functions such as `malloc` and `free`. But when no external memory allocation systems exist, such as in hardware design and freestanding software development, other techniques must be used.

One alternative approach to store the entire tree inside of a single contiguous block of memory, which can be statically allocated. This block of memory can be represented as an array of tree nodes because all of the nodes have the same size. From here it is possible to represent all of the unused elements of the array using a *free-list* [8].

A free-list is just a singly-linked list that keeps track of which of the array elements are free. To allocate a new element, simply remove the first element from the list. To free an allocated element, add it to the list. When a free-list is used to keep track of a fixed-size block of memory, the maximize size of the free-list can be known statically, allowing it to be stored in a statically allocated block of memory.

2.1.3 Hardware Tree Implementations

Tree data structures can be implemented in hardware in much the same way that they are implemented in software. The biggest differences between hardware and software implementations of trees is that the memory modules available in hardware design are much more flexible than the large unified memories used by CPUs. For example, most CPUs are limited to with a fixed word size of 32 bits, but the memory resources in FPGAs can be configured to have a variety of different word sizes [9]. This allows for the memory storing the tree data structure to be configured to have a word size that is the size of a tree node, making tree accesses more efficient. However,

because the memories in FPGAs are smaller than the memories available to CPUs, the probability of running out of memory becomes much more likely.

2.2 Recursion

A recursive algorithm is an algorithm that is defined in terms of itself. An example of this can be seen in Algorithm 2.2, which shows a recursive definition for a factorial function.

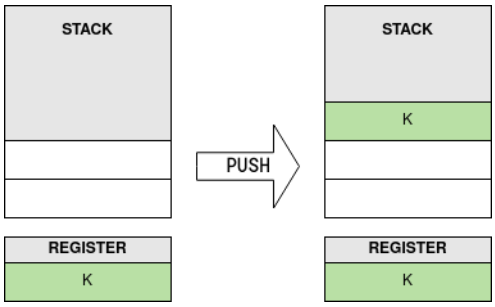
Algorithm 2.2 Recursive Factorial Function

Require: $n \geq 0$

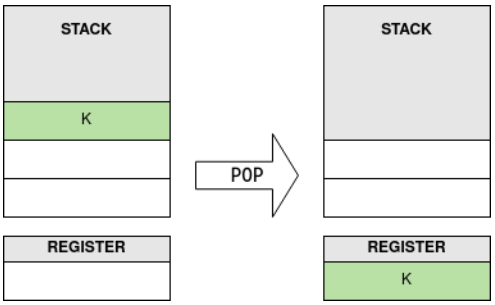
```
1: function FACT(n)
2:     if n ≤ 1 then
3:         return 1
4:     else
5:         return n * FACT(n − 1)
```

Notice that the function has two possible paths of execution depending on whether or not the argument, n, is less than or equal to 1. When $n \leq 1$, the function returns 1 without doing any additional work. This is known as the base case and it allows the function to stop calling itself and return a result. All recursive algorithms must have at least one base case or else they will never terminate. On the other hand, when $n > 1$ the recursive case is executed and the function returns $n * \text{FACT}(n-1)$. This is a recursive case because its result depends on another call to the same function, FACT. Before the recursive case can return a result it must first compute $\text{FACT}(n-1)$, causing the function to be restarted with a smaller value of n. The function will then continue calling itself with smaller and smaller values of n until the base case is reached and the function can begin returning results.

Recursion is a powerful tool because it can express repeated actions abstractly, without using explicit looping. This allows many algorithms to be written in a compact and easily understood manner, which is why recursion is supported by most



(a) Stack *push*. A *push* copies data from a register or elsewhere and places it on the top of the stack, increasing the size of the stack.



(b) Stack *pop*. A *pop* removes the data on the top of the stack and places it into a register or some other location. This decreases the size of the stack by one element.

Figure 2.2: Effects of Basic Stack Operations

high-level programming languages. However, when programs in high-level languages are translated into low-level machine code, abstract features like recursion must be given concrete implementations.

2.2.1 Low-Level Recursion

The most common implementation of recursion in low-level assembly language programs relies heavily on the use of a stack [10]. A stack is a data structure that supports two basic operations: *push* and *pop*. The *push* operation takes a value and places it on top of the stack, causing the stack to grow in size, as shown in Figure 2.2a. Inversely, the *pop* operation removes and returns the value on the top of the stack, causing the stack to shrink, which can be seen in Figure 2.2b. A stack is considered a last in, first out (LIFO) data structure because the only value that can be removed from a stack via a *pop* is the value on top of the stack, which is always the newest value.

In order to illustrate how a stack can be used to implement recursion in low-level software, Listing 2.1 shows the factorial algorithm written in the x86_64 assembly language. The recursive case of the program, which starts at the `.recur` label, has explicit `push` and `pop` instructions for using the stack. The `push` command is used to

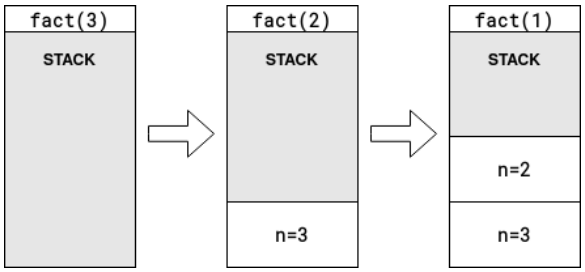
Listing 2.1: Factorial Function in x86_64 Assembly

```
1  ; fact - Factorial Function
2  ;
3  ; Registers:
4  ;   rdx - Argument
5  ;   rax - Return
6  fact:
7      cmp     rdx, 1 ; check for the base-case
8      jg      .recur ; go to .recur if (n > 1)
9      mov     rax, 1 ; set the return value to 1
10     ret            ; return to the previous call
11 .recur:
12     push    rdx    ; save n on the stack
13     sub     rdx, 1 ; n <- n-1
14     call    fact   ; start a new function call
15     pop     rdx    ; restore n from the stack
16     imul    rax, rdx
17     ret            ; return to the previous call
```

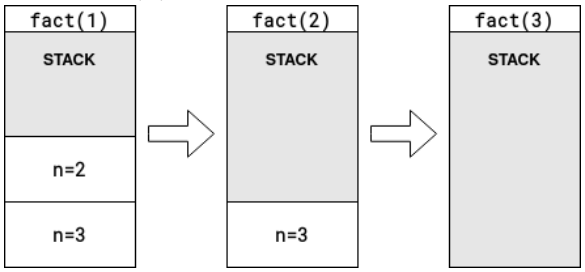
save the argument, n, on the stack while the recursive function call is executed. The `pop` operation is then used to retrieve n from the stack after the function call returns so that it can be used to compute $n * \texttt{fact}(n-1)$.

To demonstrate this process, Figure 2.3 shows the simplified state of the CPU's stack as it computes `fact`(3). Figure 2.3a shows the stack at the start of each function call and Figure 2.3b shows the stack just before each function returns. The figure demonstrates how the stack grows and shrinks dynamically as functions execute their push and pop operations. This dynamic resizing allows each function instance to ignore any changes to the stack made by subsequent function calls, since such changes are undone by the time the function instance resumes its execution.

It should be noted that because the LIFO structure of stacks works well with the ordering of function calls and returns, stacks are often used for function bookkeeping as well. For example, in Listing 2.1 the `call` instruction pushes information onto the stack that lets the CPU know where to resume execution once a call completes. The `ret` instruction pops this information off the stack and resumes execution at the specified location. This is an important use of stacks when implementing function calls



(a) Adding Recursive Calls



(b) Resolving Recursive Calls

Figure 2.3: Stack During Execution of `fact`(3)

and many hardware implementations of recursion use stacks for a similar purpose [2, 3].

2.2.2 Divide-and-Conquer Algorithms

Divide-and-conquer algorithms are a subset of recursive algorithms which take a large problem and split it into multiple recursive sub-problems. A simple example of this is the Fibonacci function, which can be seen in Algorithm 2.3. The function calculates the n^{th} Fibonacci number by summing the previous two numbers in the sequence, which are both calculated by recursively invoking the same Fibonacci function.

Algorithm 2.3 Fibonacci Function

Require: $n \geq 0$

```
1: function FIB(n)
2:     if n = 0 then
3:         return 0
4:     else if n = 1 then
5:         return 1
6:     else
7:         return FIB(n − 1) + FIB(n − 2)
```

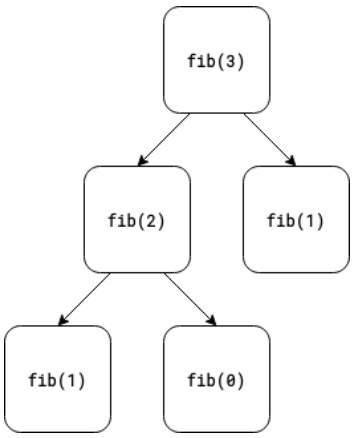


Figure 2.4: Fibonacci Call-Tree with n=3

An important characteristic of divide-and-conquer algorithms is that a dependency graph of their recursive function calls forms a tree structure [6], like the one shown in Figure 2.4. By comparison, the dependency graph of a recursive function that is not divide-and-conquer would form a straight line with no branching paths. The tree structure of the recursive function calls is interesting because the branches in the tree represent independent function calls that can be computed in parallel.

However, executing function calls in parallel makes it difficult to share data structures. For example, if multiple instances of a function were to share a stack, it would result in a data race as all function instances tried to simultaneously modify the top of the stack. This would produce inconsistent results and is the reason why stack-based recursion implementations struggle to exploit procedure-level parallelism.

There are some implementations of recursion that can parallelize these independent function calls [6, 7], but they navigate this issue by not using any shared data structures. Instead, they construct branching pipelines where all function calls are executed in parallel and each function's data is forwarded through the pipeline as needed.

2.3 Hardware Design Methodologies

In modern hardware design there are two major design methodologies: register-transfer level (RTL) design and high-level synthesis (HLS). RTL design is the traditional methodology and has been around for several decades, whereas HLS is much newer and still evolving.

In RTL design, a hardware description language (HDL) is used to describe the hardware that implements the desired functionality. HLS, on the other hand, involves designing hardware by describing the algorithm it implements using a high-level programming language such as C/C++. It is generally understood that HLS is more convenient when designing hardware that computes an algorithm, though the quality of results, in terms of size and speed, might not reach those of hand-crafted RTL designs [11, 12].

Implementing recursive algorithms in hardware is relevant to both design methodologies and there have been solutions targeted at both RTL design [2, 6] and HLS [3, 7]. Though the issue is perhaps more salient in the context of HLS, which focusses on making hardware implementations of algorithms easier. This is especially true because most current HLS tools do not support recursive functions [13, 1], which makes HLS-compatible solutions even more compelling.

2.4 VivadoHLS

One of the most popular HLS tools currently in use is the VivadoHLS tool [1], which is developed by Xilinx to work with their FPGAs. VivadoHLS uses C/C++ programs as its input, which it synthesizes into a RTL design in a HDL such as VHDL or Verilog.

```
#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

 dout_t temp;

 *sum = in1 + in2 + *sum;
 temp = in1 + in2;

 return  temp;
}
```

Figure 2.5: A Simple HLS Function [1]

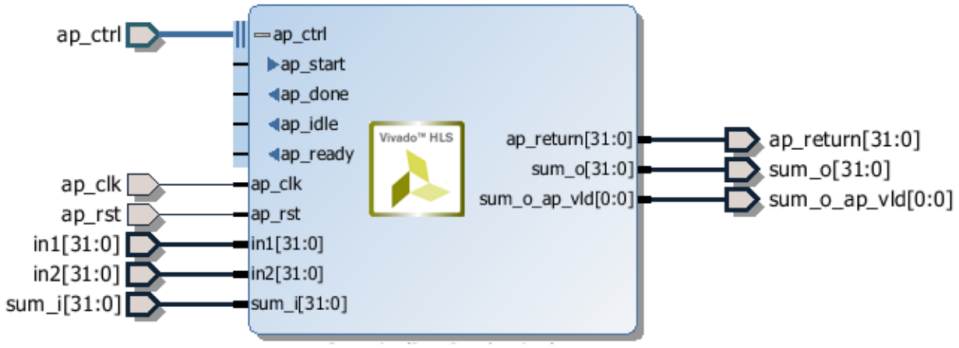


Figure 2.6: A Hardware Module Generated by VivadoHLS for Figure 2.5 [1]

2.4.1 Function Synthesis

The top-level of every VivadoHLS design is a C or C++ function, which VivadoHLS compiles into a hardware module. In order to better examine how this process works, consider the simple function shown in Figure 2.5. This function takes in three inputs (`in1`, `in2`, and the current value of `*sum`) and produces two outputs (the return value and the updated value of `*sum`). Using the default synthesis settings, this function produces a hardware module with the interface shown in Figure 2.6.

In this hardware module, there are a couple of different kinds of ports. First, there are the clock and reset ports (`ap_clk` and `ap_rst`), which can be found in most non-trivial hardware designs. Then, there are the hardware ports that correspond to the inputs and outputs of the function, such as `in1`, `sum_i`, and `ap_return`. Finally, there are the control signals (labeled as `ap_ctrl`), which are used to control the operation of the generated module using Xilinx's own protocol. The proper use of these control

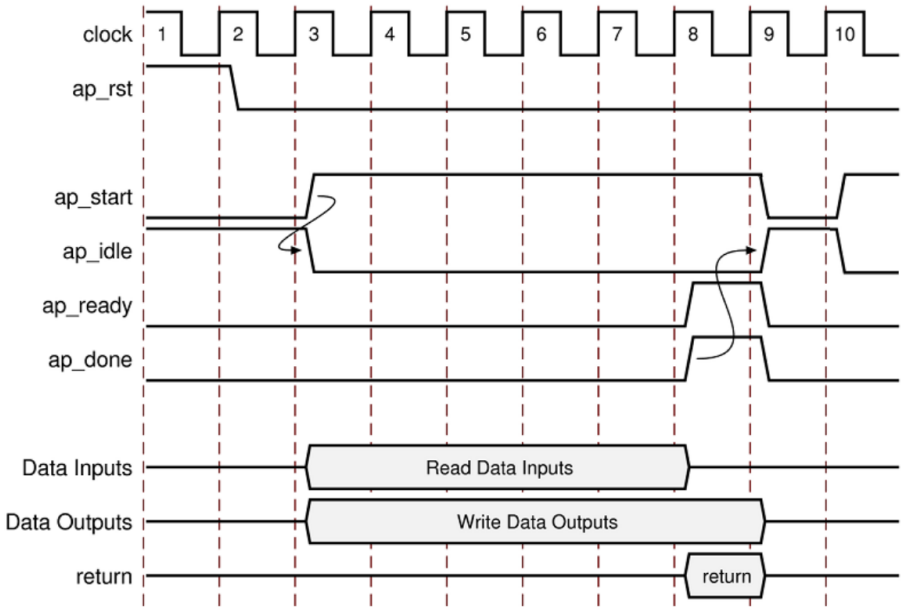


Figure 2.7: Timing Diagram for Hardware Generated by VivadoHLS [1]

signals is shown in the timing diagram shown in Figure 2.7.

To begin executing a function the `ap_start` signal is asserted high, which causes the `ap_idle` signal to go low, indicating that the function is executing. At this point, the function arguments should be held at their current value until the `ap_ready` signal goes high. The output signals from the function become valid once their corresponding `ap_vld` go high. For example, the data from `sum_o` port in Figure 2.6 will be valid when the `sum_o_ap_vld` is high. Finally, the return from the function is valid once the `ap_done` signal goes high. Using all of these signals, it is possible to control the execution of the function and know when its outputs are valid.

2.4.2 Memory Interfaces

In Figure 2.6 the function's pointer argument is synthesized into separate input and output ports since it is used as a simple input and output with no offsets or indexing. However, sometimes it is necessary for a function or program to access some external memory that stores multiple values. This can be accomplished by using an array argument to the function, which by default is synthesized into a memory port that

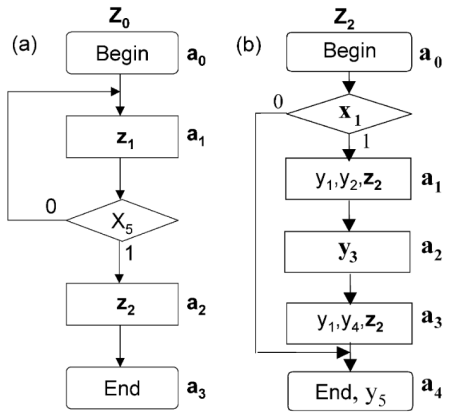


Figure 2.8: Two Examples of HFSMs
(a) has hierarchical calls and (b) has recursive calls [2]

can be attached to an external memory.

The protocol used by these ports can be set at compile time. The default protocol is Xilinx-Specific, but VivadoHLS can be set to generate memory ports that use the Advanced eXtensible Interface (AXI), which is a widely used protocol that can be used with a variety of different memories and peripherals.

2.5 Related Work

Due to the usefulness of recursive functions and their prevalence in math and computer science there have been many approaches developed over the years for synthesizing recursive functions in hardware [5]. Of these approaches, the most popular type are the stack-based approaches. These solutions implement recursion in a method similar to that used in low-level software where stack memory is used, in part, to suspend running instances of the function while they wait for their recursive calls to complete.

2.5.1 Hierarchical State Machines

Of the stack-based solutions, the most prominent are those developed by Sklyarov et al. that use hierarchical finite state machines (HFSMs) to handle recursion. This is a straightforward RTL approach that can be used to implement any recursive

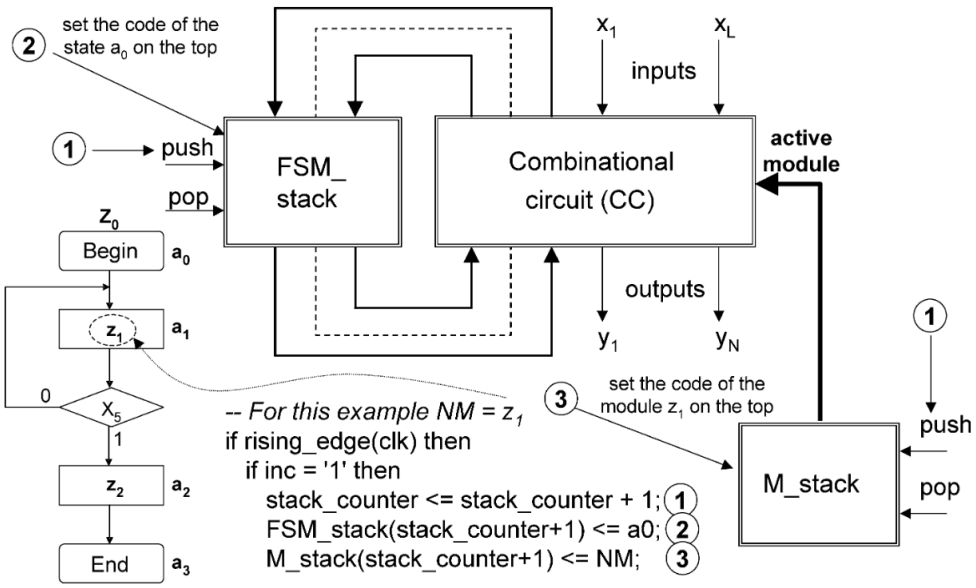


Figure 2.9: HFSM Execution System Handling a Function Call [2]

function [2, 14]. When using this approach to implement a recursive function, the first step is to generate a HFSM for the function. An example of HFSMs used by this solution can be seen in Figure 2.8, which shows two modules—one which calls other state machines and one which calls itself. In the figure, the different letters represent different types of objects: `z` is for state machines, `x` is for input actions, `y` is for output actions, and `a` is for state machine states.

Regardless of the exact form of the HFSMs, they can be executed by a two-stack system. The first stack is called the module stack (M_stack), and is used to keep track of which state machines are executing. The value on the top of this stack determines which of the state machines is being actively executed by the hardware. The second stack is the finite state machine (FSM) stack, which keeps track of the active states for all of the modules in the module stack. The value on the top of the FSM stack indicates the current state of the actively executing module, and controls the execution of the rest of the circuit. Figure 2.9 illustrates how the system handles a function call, while Figure 2.10 shows the VHDL code that defines how the stacks operate.

```
process(clk,rst)
begin
  if rst = '1' then
     stack_counter <= 0;
     FSM_stack(stack_counter) <= a0;
     M_stack(stack_counter) <= z0;
     return_flag <= '0'; error <= '0';
  elsif rising_edge(clk) then
     if inc = '1' then
        if stack_counter = stack_size then
           error <= '1';
        else
           stack_counter <= stack_counter + 1;
           FSM_stack(stack_counter+1) <= a0;
           M_stack(stack_counter+1) <= NM;
        end if;
     elsif dec = '1' then
        stack_counter <= stack_counter - 1;
        return_flag <= '1';
     else     FSM_stack(stack_counter) <= NS;
              return_flag <= '0';
     end if;
  end if;
end process;
```

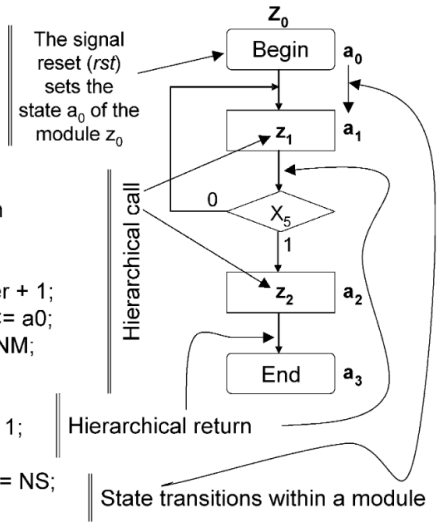


Figure 2.10: VHDL Code for Controlling the System's Stacks [2]

When a function call occurs, the stack counter is incremented in order to encode a *push* operation. The value that is pushed onto the FSM stack is `a0`, since that is the first state in all of the modules. The value that is pushed onto the modules stack is a reference to the state machine that is being called, which is the next module or `NM`. This solution is flexible because this same approach and code can be used with any HFSM.

However, because this solution is stack-based, it naturally struggles to parallelize divide-and-conquer algorithms, as mentioned in Section 2.2.2. The authors have proposed a number of solutions to mitigate the issue. First, they propose that it's possible to parallelize some of the work by creating multiple HSFM processors in a master-slave hierarchy and allowing the master to dynamically assign work to its slaves [15]. The issue with this, however, is that it is has poor load balancing and it is possible for the master processor to spend a lot of time idly waiting for the slave to complete its work.

Another approach, that the authors propose in the context of binary-tree sorting,

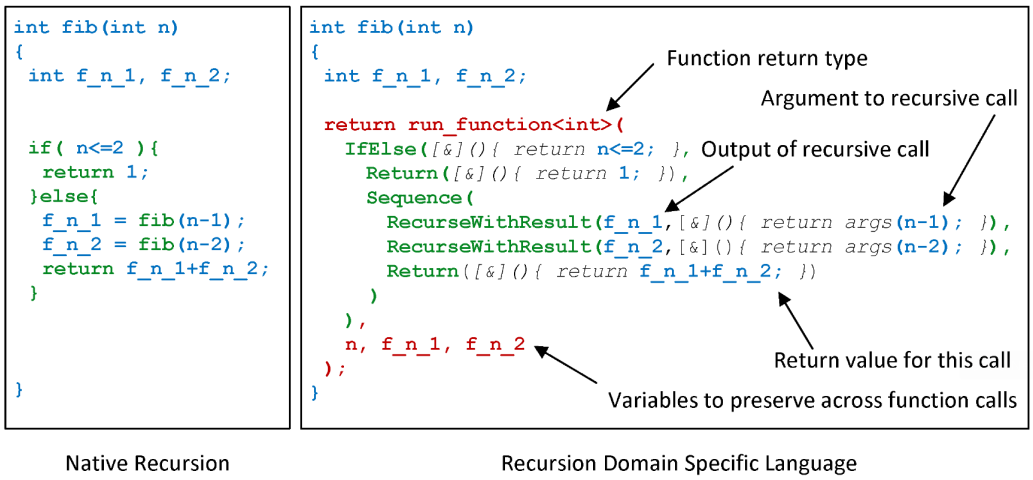


Figure 2.11: Recursion DSL Example [3]

is instantiating several HSFM processors and dividing the work up evenly between them at compile time so that each processor gets the same amount of work [15]. This solves the load balancing of the master-slave architecture, but it only works when the problem can be divided-up before runtime. This is not an issue for most sorting algorithms, but for algorithms that dynamically generate work, like the Fibonacci function in Algorithm 2.3, this condition is prohibitive.

2.5.2 HLSRecurse

An alternative stack-based recursion implementation that works with HLS tools is HLSRecurse [3]. This approach is similar to the HFSM approaches in that it constructs state machines that can be suspended to stacks. The biggest difference is that HLSRecurse automatically generates the stacks and state machines from programs written in a domain-specific language (DSL), which can save a lot of work on the user's part when compared to manually implementing state machines in an HDL. An example of this DSL can be seen in Figure 2.11, which shows the conversion of a Fibonacci function written in standard C++ to an implementation that uses the DSL.

HLSRecurse is implemented as a pure C++ library, which makes it compatible

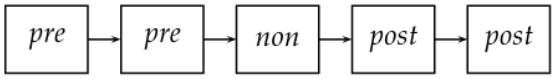


Figure 2.12: Pipeline for a Function with Two Recursive Calls [4]

with multiple HLS tools as well as standard C++ compilers. Though in order to accomplish this, it relies heavily on advanced C++ features such as templates and lambdas and requires its users to program in a functional style that is very different from most C++ code.

2.5.3 Dynamic Pipeline

Of the approaches that do not use a stack, one of the most interesting is the solution proposed by Ferizis and Gindy [6]. This approach seeks to construct an execution pipeline for divide-and-conquer algorithms that can compute independent function calls in parallel. The construction of the pipeline begins by breaking the recursive function into three pieces: the base case (non-recursive step), the part of the recursive case that occurs before the recursive calls (pre-recursive step), and the part that occurs after the recursive calls (post-recursive step). Once the different parts of the function have been identified, the pipeline is constructed as shown in Figure 2.12 with all of the pre-recursive steps followed by the non-recursive step and all of the post-recursive steps.

Because the pipeline needs a pre-recursive and post-recursive step for each level of recursion depth, the maximum depth of recursion must be known in order to construct an effective pipeline. The authors navigate this issue by dynamically computing the recursion depth of the function based on the arguments it is passed and using dynamic partial reconfiguration to add pipeline stages as needed. This has the benefit of keeping the area used by the pipeline as small as possible, but it means that this approach is only viable on devices that support dynamic partial reconfiguration, like field programmable gate arrays (FPGAs). Also, the construction of a pipeline for

executing every function call in parallel can be prohibitively expensive when design space is limited.

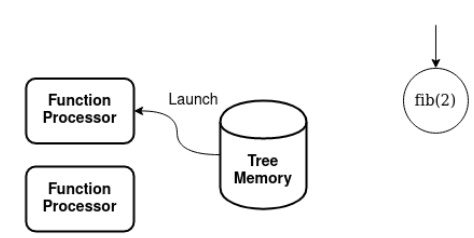
2.5.4 Recursion Flattening

An alternative solution in the same vein is the "recursion flattening" approach proposed by Stitt and Villarreal [7]. Similar to the solution proposed by Ferizis and Gindy, recursion flattening seeks to construct a pipeline for executing recursive functions, especially those with parallel recursion. However, where the Ferizis-Gindy approach relies on dynamic analysis and partial reconfiguration to ensure that the pipeline has enough stages, recursion flattening uses static analysis to determine the maximum recursion depth and constructs the maximum number of pipeline stages. This simplification allowed the authors to integrate their solution with an HLS compiler, making it much easier to use. However, the authors' use of static analysis restricts the algorithms that can be used, with some notable algorithms, like quicksort, being excluded.

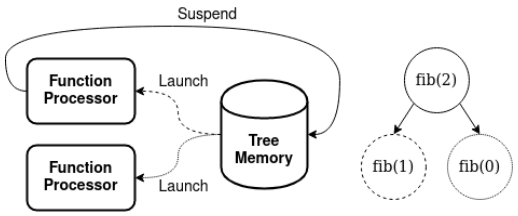
2.6 Contribution

This work presents a new solution for executing recursive algorithms in hardware, which we call TreeRecur, that can exploit the procedure-level parallelism of divide-and-conquer algorithms to a user-defined extent. By allowing for a flexible level of parallelism, TreeRecur aims to strike a middle-ground between the existing stack-based and pipelined solutions, enabling compromises between design size, energy consumption, and throughput that would have previously been unachievable.

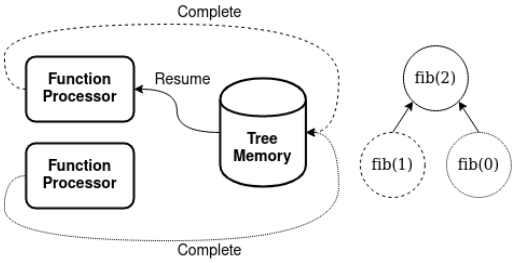
The TreeRecur approach is built using a hybrid methodology that combines both HLS and RTL design, allowing for the user-friendliness of HLS design while leveraging the precise control afforded by RTL design. This is accomplished by breaking the design into two parts, which are referred to as the front-end and the back-end. The



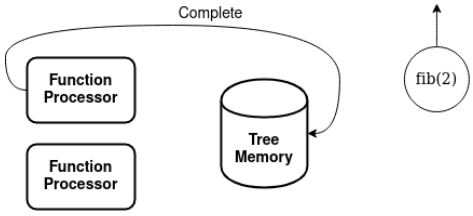
(a) `fib`(2) starts executing.



(b) `fib`(2) suspends and its child recursive calls begin execution.



(c) Execution of `fib`(1) and `fib`(0) complete execution, allowing for `fib`(2) to resume execution.



(d) `fib`(2) completes execution, returning a final result

Figure 2.13: Parallel Execution of `fib`(2)

front-end contains a configurable number of independent function processors generated with VivadoHLS for the purpose of computing independent recursive function calls in parallel. The back-end is implemented with RTL design and is responsible for keeping track of all the independently executing function calls and accomplishes this using a memory with a tree data structure, which can effectively represent the branching call structure of divide-and-conquer algorithms.

A high-level illustration of TreeRecur can be seen in Figure 2.13, which shows the computation the Fibonacci function form Algorithm 2.3 applied to an argument of 2. Along with the execution, the figure also depicts the state of the call-tree at each step of the execution. In Figure 2.13a, the execution of `fib`(2) begins and the call is sent to a processor to be executed. Since `fib`(2) executes the recursive case of the algorithm, the function call must be suspended and its two child recursive calls must be sent to the processors, which is shown in Figure 2.13b. Since both of the child recursive calls invoke base cases, they return complete results. This allows for `fib`(2) to return

to the function processors to resume its execution, as shown in Figure 2.13c. After resuming, `fib`(2) is able to complete its execution and return a final result, which is seen in Figure 2.13d.

To benchmark the effectiveness of TreeRecur, we tested it on three different divide-and-conquer algorithms: Fibonacci, quicksort, and block matrix multiplication. For each of these algorithms, execution speed, energy consumption, and hardware resource usage data was collected and the timing and energy data was compared to software implementations of the same algorithms. Where possible, multiple variations of the algorithms were tested in order examine how the variations affected performance.

Chapter 3

HLS Front End

The front-end of TreeRecur is responsible for computing the non-recursive aspects of recursive algorithms and consists of a collection of independent function processors along with some supporting hardware. VivadoHLS is used to generate the function processors, which allows for them to be written in C++, making it easier to define processors for new algorithms.

3.1 Front-End Interface

Before diving into the details of how the front-end is implemented, it is helpful to first take a high-level look at the entire system and how the pieces are connected, which can be seen in Figure 3.1. The front-end is attached to the back-end via four First-In First-Out (FIFO) queues, each of which sends commands about a different kind of action. Two of these actions are sent from the back-end to the front-end:

- **Launch:** A launch action is a request for the front-end to begin executing a new function call.
- **Resume:** A resume action is a request for the front-end to resume executing a previously suspended function call.

The other two actions go from the front-end to the back-end:

- **Suspend:** A suspend action is a request for the back-end to pause the execution

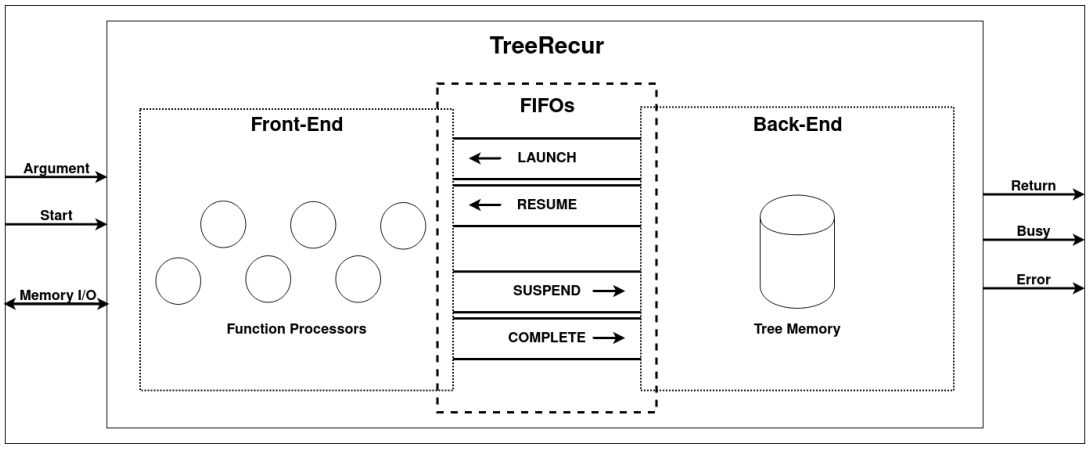


Figure 3.1: A High-Level Overview of TreeRecur
This overview shows how the connection between the front-end and back-end is comprised of four command FIFOs. It also shows the system's top-level input and output (I/O) ports, which will be discussed in more detail in Chapter 4.

of a function call while it waits for recursive calls to complete. This involves saving the call's environment data and launching its child recursive calls.

- **Complete:** A complete action is a notification to the back-end that a function call has completed execution and returned a result. The back-end is expected to save the result and resume execution of the parent function call if all of its other children have already completed.

Each FIFO has its own data type which holds the information necessary to complete the type of action associated with the queue. These data types are shown in Figure 3.2, which shows their definitions in C++. An important thing to note about these data types is that they are templates since the exact data transported depends on function-specific information, such as argument and return types.

3.1.1 HLS Interface

With the four interface actions defined, the next step is determine how to implement these actions in VivadoHLS. As mentioned in Section 2.4, VivaodHLS is a closed-source tool and its documentation provides very little information about how the internals of function synthesis are implemented. This makes it infeasible to handle

```
1 template<typename A>
2 struct LaunchType {
3     A argument;
4 };
```

(a) The data structure passed through the **launch** FIFO. The template parameter `A` is the argument type of the function and the `argument` element is the argument for the new function call being launched.

```
1 template<typename R>
2 struct CompleteType {
3     R result;
4 };
```

(b) The data structured passed through the **complete** FIFO. The template parameter `R` is the return type of the function and the `result` element is the return from the completed function call.

```
1 template<typename R, typename E, int N>
2 struct ResumeType {
3     R results[N];
4     E env;
5 };
```

(c) The data structure passed through the **resume** FIFO. The first template parameter, `R`, is the same as for the `CompleteType` structure. The other template parameters, `E` and `N`, represent an environment type and the number of recursive function calls that are made by each function instance. The `results` element stores the results from all of the child recursive calls and `env` stores the environment that was preserved when the function call was suspended.

```
1 template<typename A, typename E, int N>
2 struct SuspendType {
3     A arguments[N];
4     E env;
5 };
```

(d) The data structure passed through the **suspend** FIFO. The first template parameter, `A`, is the same as for the `LaunchType` structure and the other parameters, `E` and `N`, are the same as in the `ResumeType` structure. The `arguments` element stores the arguments for all of the child recursive calls and `env` stores the environment that must be preserved while the function call is suspended.

Figure 3.2: C++ Data types for the FIFO Elements

recursive function calls by modifying the synthesis process. To avoid this, the four interface actions are implemented using function calls and returns, whose synthesis is well defined [1].

The two actions that are triggered by the back-end, **launch** and **resume**, can be implemented as function calls and their associated data can be encoded as function arguments. Similarly, the two the front-end actions, **suspend** and **complete**, can be represented as function returns and their data can be encoded as return values.

A consequence of this approach is that recursive functions must be transformed so that recursive calls are replaced with return statements (in order to trigger **suspend** actions) and function resumptions can be represented as new function calls. The simplest version of this transformation is to split the function into two pieces: one that handles **launch** actions and one that handles **resume** actions. This can be

Listing 3.1: Factorial Function after Split-Transformation
The function has been transformed to receive interface actions as arguments and generate them as returns. The return type of `fact_launch` has been omitted since it can't be represented with the current types.

```
1  using FactLaunch = LaunchType<unsigned int>;
2  using FactResume = ResumeType<unsigned int, unsigned int, 1>;
3  using FactSuspend = SuspendType<unsigned int, unsigned int, 1>;
4  using FactComplete = CompleteType<unsigned int>;
5
6  /*unkown*/ fact_launch(const FactLaunch arg) {
7      const unsigned int n = arg.argument;
8      if (n <= 1) {
9          return FactComplete{1};
10     } else {
11         return FactSuspend{n-1, n};
12     }
13 }
14
15 FactComplete fact_resume(const FactResume arg) {
16     const unsigned int child_result = arg.results[0];
17     const unsigned int n = arg.env;
18     return FactComplete{n * child_result};
19 }
```

seen in Listing 3.1, which shows a transformed version of the factorial function from Algorithm 2.2 that uses the data types defined in Figure 3.2.

The first function, `fact_launch`, handles **launch** actions. Because launch actions represent new function calls, the argument to `fact_launch` can be converted to the argument of the original function, n, as seen on line 7 of Listing 3.1. As in the original function, if $n \leq 1$ then the base case is executed and a result of 1 is returned (via a **complete** action in this case). However, if $n > 1$ then `fact_launch` returns a **suspend** action instead of making a recursive call. The **suspend** action will cause the back-end to save the value of n, and generate a new **launch** action with an argument of $n - 1$. Unfortunately, because `fact_launch` can return either a `FactComplete` or a `FactSuspend`, its return type cannot be represented with the types that we have defined up to this point. This will be addressed in the next section.

The second function, `fact_resume`, handles **resume** actions. This function is very simple because in the original algorithm, the only action that occurs after a function call is resumed is a single multiplication.

Listing 3.2: HLS Recursion Argument Data-Type

```
1  //----------- Argument Types -----------
2  enum ArgTag {
3      LAUNCH=0,
4      RESUME=1,
5  };
6
7  template<typename A, typename R, typename E, int N>
8  struct ArgType {
9      ArgTag tag;
10     A argument;
11     R results[N];
12     E env;
13 };
```

3.1.2 Single-Function Approach

The problem with splitting the function into pieces in this way, is that it would result in VivadoHLS generating two separate function processors—one for each function. This can be avoided by merging the two functions from Listing 3.1 into a single function that takes either a `FactLaunch` or a `FactResume` as an argument and produces either a `FactSuspend` or `FactComplete` as a return.

This can be accomplished by creating two new data types: one for the argument to the function, which has the elements of both `LaunchType` and `ResumeType` and one for the return type, which contains the elements of both `SuspendType` and `CompleteType`. Additionally, these new data types need a tag to distinguish which type of interface action the structure represents. The data types developed and used in the final implementation are shown in Listing 3.2 and Listing 3.3. In these data types, the template parameters are the same as those from Figure 3.2.

Defining these custom types is useful because using consistent argument and return types for all of the HLS function processors makes it easier to control the hardware produced by VivadoHLS. This in turn makes it easier to interface the generated hardware with the rest of the system.

With this approach it is also possible to implement algorithms with multiple sets of recursive calls, such as the Ackermann function. This can be accomplished by

Listing 3.3: HLS Recursion Return Data-Type

```
1  //----------- Return Types -----------
2  enum RetTag {
3      SUSPEND=0,
4      COMPLETE=1,
5  };
6
7  template<typename A, typename R, typename E, int N>
8  struct RetType {
9      RetTag tag;
10     R result;
11     A arguments[N];
12     E env;
13 };
```

encoding information for distinguishing subsequent suspends and resumes inside of the environment type that is saved during a **suspend** action and retrieved during a **resume**.

3.2 C++ Front End

To see how the full approach can be used to implement a real divide-and-conquer algorithm, consider the Fibonacci code in Listing 3.4. The function starts off by checking the argument's `tag` flag in order to see if the function call represents the beginning of a new function instance or the resumption of an old one. If the function call is new instance resulting from a **launch** action, then the value of the function argument is checked to determine whether a base or recursive case needs to be executed. The base cases both set the return's `tag` flag to `COMPLETE` and the `result` element to the appropriate value before returning. In contrast, the recursive case sets the `tag` flag to `SUSPEND` and populates the `arguments` array with the arguments for its two child recursive calls. When the function call is the result of a **resume** action, it sets the return value to be the sum of the results of its child calls and sets the `tag` flag to `COMPLETE` before returning.

This function can be compiled by VivadoHLS to produce a function processor that is compatible with the rest of TreeRecur. The number of these function processors

Listing 3.4: C++ Front End for Fibonacci Function

```
#include "structs.h"
#include <cstdint>

using FibArg = ArgType<uint8_t, uint32_t, uint8_t, 2>;
using FibRet = RetType<uint8_t, uint32_t, uint8_t, 2>;

FibRet proc(const FibArg argVal)
{
    FibRet retVal;
    if (argVal.tag == LAUNCH) {
        uint8_t arg = argVal.argument;
        if (arg == 0) {
            retVal.tag = COMPLETE;
            retVal.result = 0;
        } else if (arg == 1) {
            retVal.tag = COMPLETE;
            retVal.result = 1;
        } else {
            retVal.tag = SUSPEND;
            retVal.arguments[0] = arg-1;
            retVal.arguments[1] = arg-2;
        }
    } else { // argVal.tag == RESUME:
        retVal.tag = COMPLETE;
        retVal.result = argVal.results[0] + argVal.results[1];
    }
    return retVal;
}
```

Listing 3.5: A simple configuration file where `NUM_PROCS` is the number of function processors to generate, `FIFO_DEPTH` is the depth of the interface FIFOs, and `MEM_SIZE` is the maximum number of calls that can be stored in the call-tree.

```
NUM_PROCS       16
FIFO_DEPTH      512
MEM_SIZE        1024
```

that will appear in the final hardware is controlled by a configuration file that is set by the user before compiling the system. An example of a simple configuration file is shown in Listing 3.5.

3.3 Front-End Architecture

With the generation of the independent function processors defined, we can examine how they fit into the architecture of the rest of the front-end. As previously mentioned, the front-end is connected to the back-end via four FIFOs—one for each interface action, which can be seen in Figure 3.3. In order to distribute the work from the back-

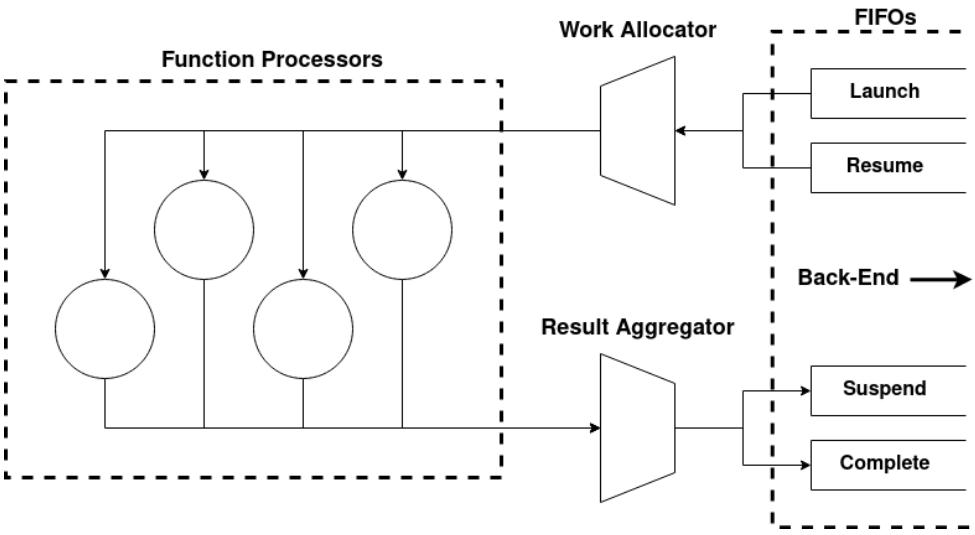


Figure 3.3: Front-End System Diagram

end across the available function processors, a custom allocator module was designed that assigns **launch** and **resume** actions to the first available processor. Similarly, the system also contains an aggregator which collects **suspend** and **complete** actions from the processors and passes them to the appropriate FIFOs.

3.3.1 Memory-Interfacing Variation

The system described so far is effective when the argument and return data for the function being computed are scalar values that can be passed through a FIFO, but is less effective when the necessary data is a vector or needs to be shared between processors. This is problematic for many algorithms, like sorting algorithms, where the data is large and sibling functions calls access disjoint regions of memory.

The solution to this problem is to allow for the function processors to access a shared memory external to the rest of the system. This memory can then be populated with data before the computation begins and modified by the processors as necessary. An alternative version of the front-end that supports this use of external memory is shown in Figure 3.4. In this figure, each of the processors have an external memory port that is connected to a single external memory via a shared memory bus controlled

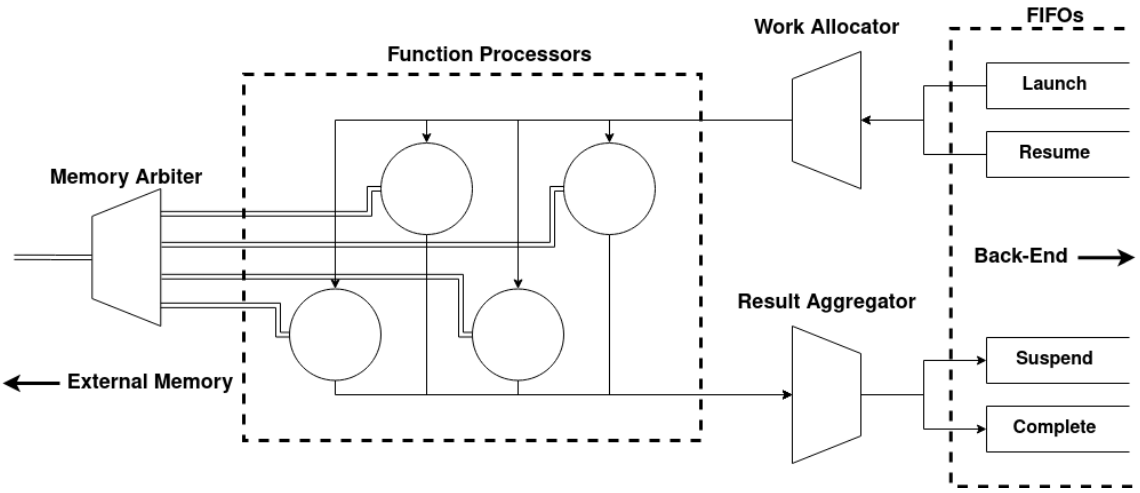


Figure 3.4: Front-End System Diagram with External Memory

by an arbiter. In VivadoHLS, an external memory ports of the function processors can be represented as an additional array argument to the function as discussed in Section 2.4.2.

Chapter 4

Tree-Based Back-End

The purpose of the back-end is to implement the aspects of recursion that are common to all recursive algorithms. Specifically, the back-end must maintain the tree data structure that keeps track of all the instances of the function that are running or suspended.

4.1 Abstract Tree Definition

In order to understand how the back-end works, it is important to first understand the tree data structure and the data that it stores. A pseudocode implementation of the data structure is shown in Algorithm 4.1. Because the exact data stored is different depending on the algorithm being implemented, the structure is shown with three template parameters: `R`, `E`, and N, which have the same meanings as they have in Figure 3.2.

Algorithm 4.1 Abstract Tree Structure

```
1: template⟨R, E, N⟩
2: structure Tree {
3:     Tree     parent
4:     int      parent_index
5:     int      active_children
6:     R[N]     child_results
7:     E        env
8: }
```

The structure has five elements—two that store information about the function call's parent, two that store information about function call's children, and one that stores data for the function call itself. First, the `parent` element is a reference to the tree node for the function instance's parent function call. This establishes the hierarchical tree structure and allows for a function call's parent to be looked up. Next, the `parent_index` element is an index for the parent tree node's `child_results` array. All of a node's children must have different values for their `parent_index`. The `active_children` element is used to keep track how many of the tree node's children have yet to return. When this value reaches 0, then the function is ready to resume. The `child_results` element stores the results of all of the function's child recursive calls. Whenever one of the function's children **complete** they store their result in the element of the function's `child_results` array that corresponds with their `parent_index`. Finally, the `env` element stores the data saved when a function is suspended and is passed back to the front-end with the `child_results` element when the function call **resumes**.

The way that this structure can be used to implement the **suspend** interface action is shown in Algorithm 4.2. As described in Section 3.1, when a **suspend** action happens the front-end passes the back-end arguments for child recursive calls and environment data that will be needed when the function resumes. The back-end is expected to **launch** new recursive calls for all the arguments it is passed and save the environment data until the function is ready to **resume**.

Algorithm 4.2 Function Suspend Pseudocode

```
1: function SUSPEND(Tree node, A[N] calls, E env)
2:     node.env ← env
3:     for i ∈ 0..N − 1 do
4:         Tree child_node ← ALLOC_NODE( )
5:         child_node.parent ← node
6:         child_node.parent_index ← i
7:         LAUNCH(child_node, calls[i])
8:     node.active_children ← N
```

The function in Algorithm 4.2 has three arguments: the tree-node for the function being suspended and the arguments and environment data from the front-end. The function starts off by saving the environment data to the tree-node. The function then iterates through all of the child calls it needs to make, allocates a new tree-node for each function call, sets the new node's `parent` and `parent_index`, and launches a new recursive call with the appropriate argument. Finally, the function sets the node's `active_children` to be the number of child calls launched since all of the launched children are now active.

Algorithm 4.3 shows the pseudocode for how **complete** actions can be handled using the tree data structure. This involves updating the parent function call's tree-node

Algorithm 4.3 Function Complete Pseudocode

```
1: function COMPLETE(Tree node, R ret)
2:     Tree parent ← node.parent
3:     parent.child_results[node.parent_index] ← ret
4:     parent.active_children ← parent.active_children−1
5:     if parent.active_children = 0 then
6:         RESUME(parent, parent.child_results, parent.env)
7:     FREE_NODE(node)
```

with the result from the function call, decrementing the parent's `active_children` count, and resuming the parent function call if it has no active children left. Lastly, the tree-node for the completed function call is freed so that it can be reused later.

4.2 Function Call Tracking

The SUSPEND and COMPLETE functions defined in Algorithm 4.2 and Algorithm 4.3 both require a tree-node as their first argument. In order to satisfy this argument and compute these functions there must be a way to lookup the relevant tree-node whenever a **suspend** and **complete** action occurs. This is accomplished by assigning each function instance a unique identifier when its tree-node is allocated. The identifier is passed to the front-end during **launch** and **resume** actions and returned by

the front-end during **suspend** and **complete** actions. This way, when the back-end receives an interface action from the front-end it can know which function instance is responsible and lookup the relevant tree-node, which can be used to perform the necessary operations.

However, this still requires a way to lookup a tree-node based on it's identifier. This could be accomplished by storing a function instance's identifier in its tree-node and searching the tree for the right identifier every time a **suspend** or **resume** action occurs, but this approach is slow. A much simpler approach is to re-use the memory address of each tree-node as the corresponding function call's identifier. This way, a tree-node can be looked-up directly from its identifier whenever an interface action occurs, avoiding the need to perform any kind of search.

4.3 Dynamic Tree-Node Management

The pseudocode functions in Algorithm 4.2 and Algorithm 4.3 use two dynamic memory functions ALLOC_NODE and FREE_NODE, which allocate and free tree-nodes respectively. These functions are necessary because the call tree dynamically grows and shrinks as the function executes, as seen in Figure 2.13.

Because all of the tree-nodes are the same size, the dynamic memory management can be efficiently handled using an array-based tree implementation as mentioned in Section 2.1.2. A dedicated memory management module is used to keep track of which memory blocks are available using a free-list. This module is responsible for allocating the tree-nodes needed by Algorithm 4.2 and freeing the nodes provided by Algorithm 4.3. It is also responsible for raising an error if the system runs out of memory, which occurs when an allocate occurs and the free-list is empty. If this happens, the entire system stops executing the function and raises an error flag to indicate to the user that the computation failed.

The free-list could be stored in the same memory as the tree structure, using the

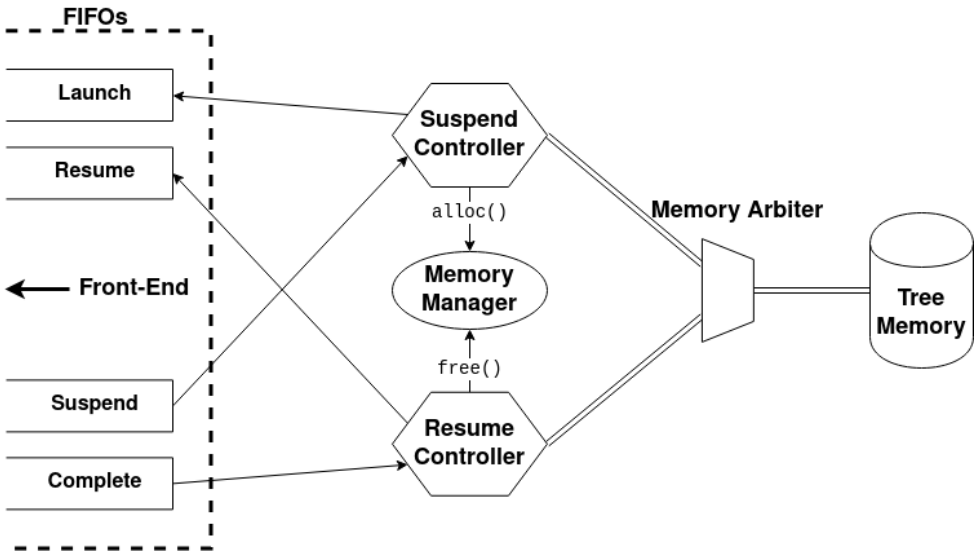


Figure 4.1: Back-End System Diagram

unallocated tree nodes as elements of the free-list. This reduces the memory usage of the system, but increases the number of accesses to the tree memory and complicates the control logic. Instead, the free-list is stored in it's own memory that's sized to have the same number of elements as the tree memory, which is the maximum size of the free-list.

4.4 Back-End Architecture

With the tree data structure and the necessary operations defined, it is possible to examine the system architecture of the back-end, which is shown in Figure 4.1. As mentioned in Section 3.3, the back-end is connected to the front-end via four FIFOs that represent the four interface actions. Aside from the memory that stores the tree data structure, the back-end has three important modules: the suspend controller, the resume controller, and the memory manager.

The suspend controller is responsible for implementing the SUSPEND function defined in Algorithm 4.2. Similarly, the resume controller implements the COMPLETE function from Algorithm 4.3, which can trigger **resume** actions. Finally, the mem-

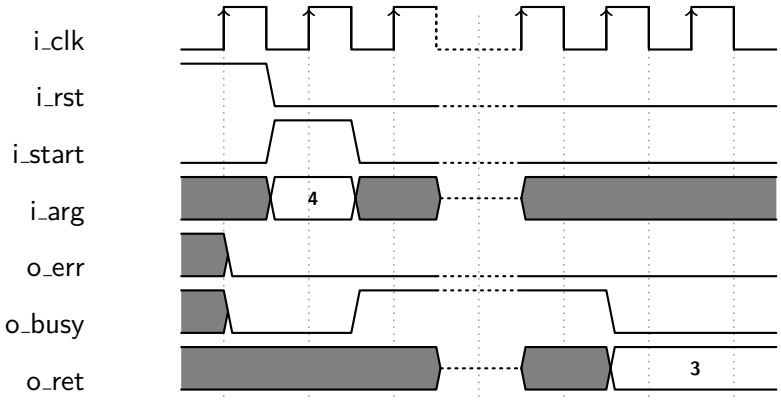


Figure 4.2: A top-level timing diagram showing the execution of `fib(4)`. The argument, `i_arg`, is latched-in when `i_start` goes high and the output, `o_ret` becomes valid when `o_busy` goes low. `o_err` indicates the presence of an error condition.

ory manager handles the dynamic allocation and freeing of tree-nodes described in Section 4.3.

4.5 Top-Level Interface

One final part of the system that is important to discuss is the top-level I/O ports of the system, which were briefly shown in Figure 3.1. The behavior and timing of these ports is shown in Figure 4.2, which depicts the execution of `fib(4)`.

Execution begins when the `i_start` signal goes high, which causes the root node of the call tree to be initialized and a **launch** action to be sent to the front-end. The argument that is sent with the **launch** is the current value of the `i_arg` port. The system then executes as previously described until the back-end receives a **complete** action for the root node of the call tree. When this happens, the `o_busy` port is driven low and the `o_ret` port is set to the value returned with the **complete** action.

The top-level ports also include an error port, `o_err`. This port is set to a nonzero value when the system encounters an error, at which point execution stops and the `o_busy` signal is driven low. There are two possible errors that can be reported by the `o_err` port:

- Out of Memory—which occurs when the dynamic tree node allocator is unable to allocate a new node. This can be fixed by increasing the size of the tree memory
- Deadlock—which is triggered when the back-end is unable to write to one of the action FIFOs for a full second. This can be fixed by increasing the size of the FIFOs.

There is also an extended version of this interface that also includes an AXI4 memory mapped master port, which is used to connect to the optional external memory that can be used by the front-end, as described in Section 3.3.1. This port lets TreeRecur be used with a wide variety of different memory modules, making it very flexible.

Chapter 5

Results

To ensure the generality of the developed solution, three different divide-and-conquer algorithms were implemented using TreeRecur. These implementations were then compared to equivalent software programs in order to evaluate the performance of the framework.

The HLS processors of the TreeRecur implementations were synthesized using VivadoHLS version 2019.1. The output of VivadoHLS was combined with the rest of the project and synthesized and simulated in Vivado 2019.1. The timing data was collected using a post-implementation simulation with a Virtex-7 XC7VX690T FPGA as the synthesis target and a system clock speed of 100 MHz.

For the software results, timing data was collected on a Intel i7-7800X processor running at 3.50 GHz with 16 GB of RAM. The code was compiled with GCC version 4.8.5 with the default optimization level. All of the software algorithms were implemented in a way that involved no multithreading or parallelism, but were otherwise kept as similar to the HLS programs as possible. The computations were timed using the high-resolution clocking features of C++'s `std::chrono` library, with the results being averaged over a million runs in order to increase their precision.

In order to ensure a fair comparison between both approaches, the different implementations of the algorithms were tested using the same sets of input data. The data used was generated using C's `rand` function, seeded with a fixed value of `0xFEED`. For

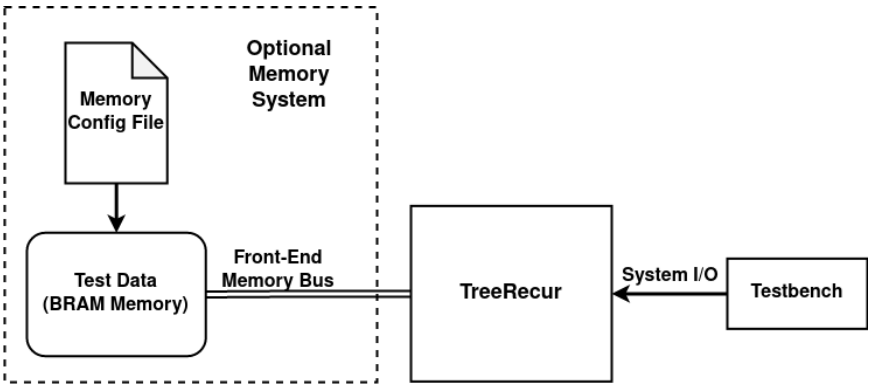


Figure 5.1: TreeRecur Testing Setup
The system's I/O ports are driven by a testbench, while the testing data is populated with a configuration file before the simulation, if necessary.

the software algorithms, this data was generated dynamically at run-time before running the tests. For the hardware implementation, the data was generated beforehand and used as the initialization data for the external memory connected to the front-end processors. The external memory used was a BRAM-based memory, generated using Xilinx IP. This configuration can be seen in Figure 5.1.

The hardware area results for the TreeRecur implementations were collected from the post-implementation utilization report generated by Vivado. Data was collected for the number of Look-Up Tables (LUTs), registers, and Block Random-Access Memories (BRAMs) used. The LUT and register utilization data was taken from section 1 of the report, while BRAM usage was taken from section 3. All of the systems were built and tested with the same system FIFO depth of 512 elements and the same tree memory size of 1024 elements.

Energy consumption for the hardware was calculated by multiplying the execution times by the total power draw reported by Vivado in the post-routing power reports. With software, it is more difficult to get an accurate measurement of energy consumption. The method used in this work involved using the Processor Counter Monitor (PCM) library [16], which is supported by Intel [17] and includes utilities for measuring energy consumption. As with the timing measurements, the power

Table 5.1: Fibonacci Execution Times (μs)

Software	1 Processor	2 Processors	4 Processors	8 Processors
4.3527	638.02	635.25	635.25	635.25

measurements were averaged over a million runs to increase precision.

5.1 Algorithm Selection

The three algorithms used to test the implementation of TreeRecur were the Fibonacci function, quicksort, and a block matrix multiplication algorithm. All of these algorithms were chosen because they are simple enough that their implementations could be written and tested in a reasonable amount of time.

These algorithms are also able to be computed using only statically allocated memory, which is important since TreeRecur currently doesn't support any kind of dynamic memory allocation. Lastly, these algorithms have significant differences in their argument and return types as well as in the number of recursive function calls they generate. This helps test the generality of TreeRecur, ensuring that it can handle a variety of different algorithms.

5.2 Fibonacci

The first algorithm used to test TreeRecur was the Fibonacci function, which was introduced in Algorithm 2.3. Because the Fibonacci function generates $O(2^n)$ function calls, the input size was restricted to be a four-bit integer in order to keep the maximum number of calls below the 1024-call limit enforced by the tree memory size used. The timing results were collected for the function run with the maximum argument of 15, and the results are shown in Table 5.1 and Table 5.2.

The timing data collected was measured in both seconds and in the number of clock cycles. The timing data measured in seconds is interesting because it is the

Table 5.2: Fibonacci Execution Times (Clock Cycles)

Software	1 Processor	2 Processors	4 Processors	8 Processors
15,234	63,802	63,525	63,525	63,525

Table 5.3: Fibonacci Energy Consumption (μJ)

Software	1 Processor	2 Processors	4 Processors	8 Processors
147.59	215.65	215.99	215.99	217.89

performance metric with the most relevance to end-users. The execution speed measured in clock cycles is also interesting since accounts for the fact that the CPU's clock frequency is 35 times faster than the FPGA's.

However, even when difference in clock speed is accounted for, the TreeRecur is still 4.17–4.19 times slower than the software implementation. This is much better than the absolute execution time where TreeRecur is between 146 and 147 times slower than the software speed, but there is still a significant difference in performance.

A likely cause of this performance gap is the greater function call overhead associated with the tree-based approach to recursion compared to stack-based solutions. As previously discussed, adding data to a stack is very straightforward, whereas the process of adding nodes to a tree is more involved. In the case of the Fibonacci function, the cost of the greater function call overhead is very noticeable since the Fibonacci function involves little computational work aside from launching function calls.

What is more interesting is the fact that the execution speed barely increases when a second processor is added and then doesn't change as more processors are added to the system. The reason for this is that the Fibonacci function calls are very simple and can be computed very quickly while the back-end **suspend** and **complete** actions are more complex and take longer to execute. This causes the back-end processing to act as a bottleneck for the system, making the addition of more processors irrelevant.

Table 5.4: Fibonacci Hardware Area

Resource	1 Processor	2 Processors	4 Processors	8 Processors
LUTs	616	676	777	980
Registers	1,132	1,258	1,506	1,998
Block RAMs	6	6	6	6

Table 5.5: Fibonacci Hardware Area

	LUTs	Registers	Block RAMs
Virtex-7 XC7VX690T	433,200	866,400	1470

Another performance statistic that can be considered is energy consumption, which is shown in Table 5.3. When looking at energy consumption, TreeRecur is only between 1.46 and 1.48 times worse than the software implementation. The energy results are impressive because although the TreeRecur implementation runs for more than 100 times longer than the software implementation, the energy consumptions are relatively close. This is the result of the small size of the hardware needed to compute the Fibonacci function, which causes its total power draw to be less than 0.5 W.

The area usage of TreeRecur with different numbers of processors is shown in Table 5.4. Using a linear regression it was found that each new processor adds about 52 LUTs and 124 registers to the area usage. Extrapolating this data backwards, it can be estimated that the system without any processors would consume about 569 LUTs, 1010 registers, and 6 BRAMs. For comparison, the total number of resources available on the FPGA used are listed in Table 5.5. From this table it can be seen that even the 8-processor system uses less than half a percent of any of the FPGA's resources.

5.3 Quicksort

The second algorithm examined was the quicksort algorithm, which is a popular sorting algorithm where a list is sorted relative to a single element of the list, called the pivot. This results in two sub-lists: one which contains all the elements less than the pivot and one that contains all the elements greater than the pivot. These two lists can then be sorted recursively in the same manner until the entire list is sorted.

Algorithm 5.1 Quicksort

```
 1: function PARTITION(arr, n)
 2:     sorted ← arr
 3:     pivot ← arr[n−1]
 4:     for i in 0 .. (n − 1) do
 5:         current ← arr[i]
 6:         if current < pivot then
 7:             temp ← *sorted
 8:             *sorted ← current
 9:             arr[i] ← temp
10:             sorted ← sorted + 1
11:     arr[n−1] ← *sorted
12:     *sorted ← pivot
13:     return sorted
14:
15: function QUICKSORT(arr, n)
16:     if n ≤ BASE_SIZE then
17:         for i in 0 .. (n − 1) do
18:             key ← arr[i]
19:             j ← i − 1
20:             while j ≥ 0&&arr[j] > key do
21:                 arr[j + 1] ← arr[j]
22:                 j ← j − 1
23:             arr[j + 1] ← key
24:     else
25:         new_arr ← PARTITION(arr, n)+1
26:         num_sorted ← new_arr − arr
27:         QUICKSORT(arr, num_sorted−1)
28:         QUICKSORT(new_arr, n−num_sorted)
```

Algorithm 5.1 shows a slight variation of quicksort where once the list is below

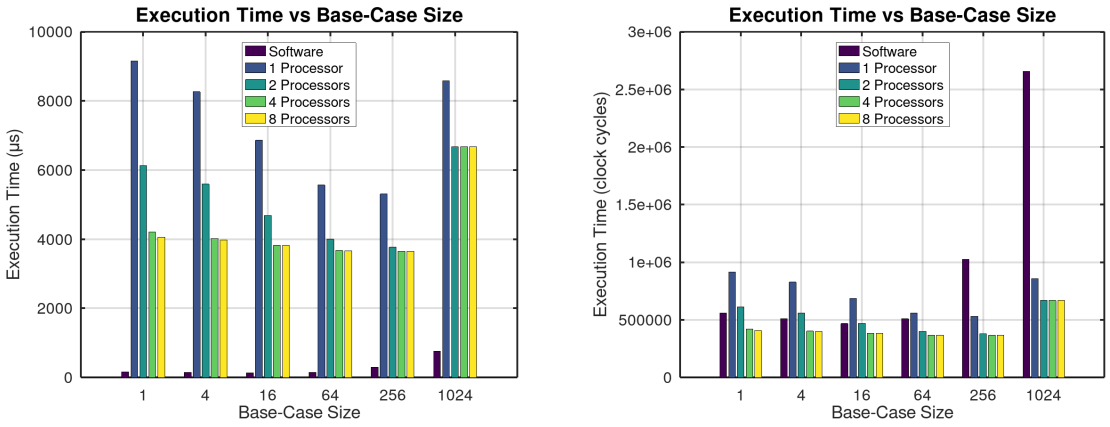
Table 5.6: Quicksort Execution Times (μs)

Base Size	Software	1 Processor	2 Processors	4 Processors	8 Processors
1	160.030	9,148.07	6,133.02	4,205.22	4,059.29
4	145.463	8,263.57	5,591.86	4,010.71	3,967.89
16	133.370	6,857.82	4,684.33	3,819.45	3,820.92
64	145.184	5,564.97	3,995.89	3,668.92	3,666.97
256	292.775	5,304.88	3,763.64	3,651.64	3,650.76
1024	759.312	8,580.88	6,673.17	6,673.17	6,673.17

a certain size, it is sorted using insertion sort, a relatively fast non-recursive sorting algorithm. The purpose of this is to increase the amount of processing that is done in many of the function calls and reduces the total number of function calls. The goal of this is to mitigate the issue seen with the Fibonacci function where the front-end processes function calls too quickly, causing the back-end to bottleneck performance. With this algorithm it is possible to control the amount of processing that is done in a function call by changing the input size that triggers the base-case, allowing the effects of the base-case size on execution speed to be examined.

The timing results for this algorithm were collected for an input size of 2,048 elements because when testing with larger input sizes, some of the simulations began to crash from a lack of memory. The timing results for the quicksort function are shown in Table 5.6 and Figure 5.2.

As with the Fibonacci algorithm, the TreeRecur implementation performs significantly worse than the software in terms of absolute execution times. However, with quicksort, many of the hardware implementations are all between 57.2 and 8.79 times slower than their software counterparts as whereas the Fibonacci hardware implementation were between 146 and 147 times slower. Also, when looking at the clock-relative timing, some of the TreeRecur implementations are able to out-perform the software for all base-case sizes. With the 1024-element base-case, the TreeRecur implementations are able to outperform the software by up to 3.98 times. Furthermore, when looking at energy consumption, we can see that the TreeRecur can outperform



(a) Quicksort execution times measured in μs.

(b) Quicksort execution times measured in device clock cycles.

Figure 5.2: Quicksort Execution Times

Table 5.7: Quicksort Energy Consumption (μJ)

Base Size	Software	1 Processor	2 Processors	4 Processors	8 Processors
1	5,881.53	3,522.01	2,379.61	1,732.55	1,891.63
4	5,434.99	3,214.53	2,275.89	1,792.78	2,102.98
16	5,012.91	2,660.83	1,845.63	1,634.72	1,937.21
64	5,500.41	2,125.82	1,626.33	1,621.66	1,888.49
256	11,588.0	2,074.21	1,497.93	1,588.46	1,865.54
1024	30,846.0	3,337.96	2,749.35	3,016.27	3,456.70

the software implementation by consuming up to 11.2 times less energy, as shown in Table 5.7.

Looking at Figure 5.2a and Figure 5.3, it can be seen that increasing the size of the base-case generally improves both the execution speed and the energy consumption. Also, the addition of more processors can significantly improve performance and energy consumption. The multi-processor systems can have execution times that are up to 2.25 times faster than the single-processor systems and energy consumptions that are up to 2.03 times lower.

The area usage data for the quicksort implementation with a 64-element base-case is shown in Table 5.8. Again, we can use a linear regression to get an approximate cost for each additional processor, which shows that each processor adds about 1,527

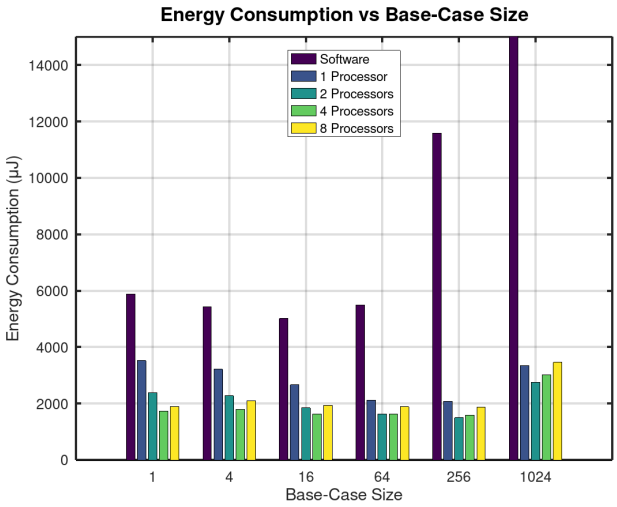


Figure 5.3: Quicksort energy consumption. The software energy consumption for the 1024-element base-case is cropped to improve readability.

Table 5.8: Quicksort Area Usage with 64-Element Base-Case Size

Resource	1 Processor	2 Processors	4 Processors	8 Processors
LUTs	2,269	3,966	6,974	13,024
Registers	3,765	6,283	10,954	20,283
Block RAMs	14.5	16	19	25

LUTs, 2,351 registers, and 1.5 BRAMs. We can also estimate that the base system without any processors would consume 830 LUTs, 1,505 registers, and 13 BRAMs.

This means that the base system for the quicksort algorithm is slightly larger than the base system used for the Fibonacci function, which has several causes. First, the quicksort algorithm uses an external memory, which consumes additional BRAMs and requires more logic to perform tasks such as arbitration of the shared bus. Also, the data structures used by the quicksort algorithm are larger than those used by the Fibonacci function, since the argument to quicksort is two 32-bit numbers, whereas the argument to the Fibonacci function was a single 4-bit number. The size of the data structures is important because larger data structures means that all of the hardware that operates on the data must be larger as well, leading to increased resource usages.

5.4 Divide-and-Conquer Matrix Multiplication

The final algorithm examined was a simple divide-and-conquer matrix multiplication algorithm. This algorithm can be used to multiply two matrices of size $2^n \times 2^n$, where $n \in \mathbb{N}$, by performing eight $2^{n-1} \times 2^{n-1}$ multiplications[8]. For two $2^n \times 2^n$ matrices, $\mathbf{A}$ and $\mathbf{B}$, the algorithm starts by breaking both matrices into four sub-matrices of size $2^{n-1} \times 2^{n-1}$, as shown in (5.1).

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_1 & \mathbf{A}_2 \\ \mathbf{A}_3 & \mathbf{A}_4 \end{pmatrix},\ \mathbf{B} = \begin{pmatrix} \mathbf{B}_1 & \mathbf{B}_2 \\ \mathbf{B}_3 & \mathbf{B}_4 \end{pmatrix} \tag{5.1}$$

The algorithm then proceeds to perform the eight multiplications and four additions shown in (5.2).

$$\begin{gathered} \mathbf{AB} = \mathbf{C} = \begin{pmatrix} \mathbf{C}_1 & \mathbf{C}_2 \\ \mathbf{C}_3 & \mathbf{C}_4 \end{pmatrix} \\ \mathbf{C}_1 = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_3;\ \mathbf{C}_2 = \mathbf{A}_1\mathbf{B}_2 + \mathbf{A}_2\mathbf{B}_4 \\ \mathbf{C}_3 = \mathbf{A}_3\mathbf{B}_1 + \mathbf{A}_4\mathbf{B}_3;\ \mathbf{C}_4 = \mathbf{A}_3\mathbf{B}_2 + \mathbf{A}_4\mathbf{B}_4 \end{gathered} \tag{5.2}$$

This is very similar to the more popular Strassen matrix multiplication algorithm, which only involves seven matrix multiplications[8]. The Strassen algorithm works by constructing ten $2^{n-1} \times 2^{n-1}$ matrices shown in (5.3).

$$\begin{gathered} \mathbf{S}_1 = \mathbf{B}_2 - \mathbf{B}_4;\ \mathbf{S}_2 = \mathbf{A}_1 + \mathbf{A}_2 \\ \mathbf{S}_3 = \mathbf{A}_3 + \mathbf{A}_4;\ \mathbf{S}_4 = \mathbf{B}_3 - \mathbf{B}_1 \\ \mathbf{S}_5 = \mathbf{A}_1 + \mathbf{A}_4;\ \mathbf{S}_6 = \mathbf{B}_1 + \mathbf{B}_4 \\ \mathbf{S}_7 = \mathbf{A}_2 - \mathbf{A}_4;\ \mathbf{S}_8 = \mathbf{B}_3 + \mathbf{B}_4 \\ \mathbf{S}_9 = \mathbf{A}_1 - \mathbf{A}_3;\ \mathbf{S}_{10} = \mathbf{B}_1 + \mathbf{B}_2 \end{gathered} \tag{5.3}$$

These matrices are used to compute the final matrix through the multiplications from (5.4) and the additions from (5.5).

$$\begin{aligned} \mathbf{P}_1 = \mathbf{A}_1\mathbf{S}_1;\ \mathbf{P}_2 = \mathbf{S}_2\mathbf{B}_4 \\ \mathbf{P}_3 = \mathbf{S}_3\mathbf{B}_1;\ \mathbf{P}_4 = \mathbf{A}_4\mathbf{S}_4 \\ \mathbf{P}_5 = \mathbf{S}_5\mathbf{S}_6;\ \mathbf{P}_6 = \mathbf{S}_7\mathbf{S}_8 \\ \mathbf{P}_7 = \mathbf{S}_9\mathbf{S}_{10} \end{aligned} \tag{5.4}$$

$$\begin{aligned} \mathbf{C}_1 = \mathbf{P}_5 + \mathbf{P}_4 - \mathbf{P}_2 + \mathbf{P}_6;\ \mathbf{C}_2 = \mathbf{P}_1 + \mathbf{P}_2 \\ \mathbf{C}_3 = \mathbf{P}_3 + \mathbf{P}_4;\ \mathbf{C}_4 = \mathbf{P}_5 + \mathbf{P}_1 - \mathbf{P}_3 - \mathbf{P}_7 \end{aligned} \tag{5.5}$$

The issue with the Strassen algorithm is that all of its intermediate matrices from (5.3) must be stored in processor accessible memory so that they can be used as arguments to recursive function calls, but the developed solution has no way to dynamically allocate memory for these matrices. The only memories available in TreeRecur are the tree memory, which can't be accessed by the processors and the optional shared external memory, which has no centralized controller that could be used to allocate memory for intermediate data. This means that in order to implement the Strassen algorithm, the memories for all of the intermediate matrices for all of the recursive calls would have to be allocated statically, which would be difficult. This is one of the major shortcomings of the version of TreeRecur because it makes it difficult or impossible to implement algorithms that generate intermediate data, unless that data is small enough to be passed around as arguments or stored in the tree memory.

However, this problem is manageable with the basic divide-and-conquer matrix multiplication algorithm because the only intermediate matrices that it needs to store are the eight matrix sub-products that result from the recursive calls. This allows for the multiplication of two $2^n \times 2^n$ matrices to be computed inside of a single statically-

Table 5.9: Matrix Multiplication Execution Times (μs)

Base Size	Software	1 Processor	2 Processors	4 Processors	8 Processors
1x1	50.697	2,912.12	2,939.43	2,939.43	2,939.43
2x2	24.196	1,238.97	894.82	587.86	481.29
4x4	17.643	544.57	413.44	296.92	259.06
8x8	15.161	266.38	206.29	171.67	159.76
16x16	14.311	138.18	156.26	156.26	156.26

allocated $2^n \times 2^n \times 2^n$ tensor. This works because at each step of execution the tensor can be divided into eight $2^{n-1} \times 2^{n-1} \times 2^{n-1}$ tensors that can be used to compute the eight $2^{n-1} \times 2^{n-1}$ sub-multiplications. The process can be repeated until the matrices being multiplied are 1×1 matrices, which can be trivially computed inside a $1 \times 1 \times 1$ tensor by performing a single multiplication.

Similar to quicksort, the divide-and-conquer matrix multiplication algorithm can have a flexibly-sized base case, which can be used to avoid the Fibonacci function's bottlenecking issue. With the matrix multiplication algorithm, a simple iterative algorithm can be used to compute matrix multiplications below a certain size, as seen in Algorithm 5.2. The matrix multiplication algorithm was tested with five different base-case sizes ranging from a 1×1 matrix multiplication to a 16×16 matrix multiplication. These implementations were tested on a 16×16 matrix multiplication and the execution times were recorded in Table 5.9 and plotted in Figure 5.4.

From the data, it can be seen that the TreeRecur implementations are between 58.0 and 9.66 times slower than the software implementation when measured in seconds. This is very similar to the quicksort algorithm, which was between 57.2 and 8.79 slower than the software programs. Thus, when looking at clock-relative timing, the TreeRecur implementations are again able to outperform the software implementation. Looking at the energy consumptions in Table 5.10 and Figure 5.5, it can be seen that, as with quicksort, the TreeRecur implementations are generally more energy efficient than the software implementation—consuming up to 8.65 times less

Algorithm 5.2 Block Matrix Multiplication

```
1: function MULT(A, B, C, n, log_stride)
Require: A and B point to 2^n × 2^n matrices
Require: C points to a 2^n × 2^n × 2^n tensor
Require: log_stride = n for the initial function call.
2:     stride ← 2^log_stride
3:     if n ≤ BASE_SIZE then
4:         offset ← 2^n
5:         for i in 0 .. (offset − 1) do
6:             for j in 0 .. (offset − 1) do
7:                 C[j + i ∗ stride] = 0
8:                 for k in 0 .. (offset − 1) do
9:                     C[j + i ∗ stride] += A[k + i ∗ stride] ∗ B[j + k ∗ stride]
10:    else
11:        new_n ← n − 1
12:        new_offset ← 2^new_n
13:        horizontal_offset ← new_offset
14:        vertical_offset ← new_offset ∗ stride
15:        depth_offset ← new_offset ∗ stride^2
16:
17:        A_1 ← A
18:        A_2 ← A + horizontal_offset
19:        A_3 ← A + vertical_offset
20:        A_4 ← A + horizontal_offset + vertical_offset
21:
22:        B_1 ← B
23:        B_2 ← B + horizontal_offset
24:        B_3 ← B + vertical_offset
25:        B_4 ← B + horizontal_offset + vertical_offset
26:
27:        C_1 ← C
28:        C_2 ← C + horizontal_offset
29:        C_3 ← C + vertical_offset
30:        C_4 ← C + horizontal_offset + vertical_offset
31:        C_5 ← C_1 + depth_offset
32:        C_6 ← C_2 + depth_offset
33:        C_7 ← C_3 + depth_offset
34:        C_8 ← C_4 + depth_offset
```

```
35:     MULT(A_1, B_1, C_1, new_n, log_stride)
36:     MULT(A_2, B_3, C_5, new_n, log_stride)
37:     MULT(A_1, B_2, C_2, new_n, log_stride)
38:     MULT(A_2, B_4, C_6, new_n, log_stride)
39:     MULT(A_3, B_1, C_3, new_n, log_stride)
40:     MULT(A_4, B_3, C_7, new_n, log_stride)
41:     MULT(A_3, B_2, C_4, new_n, log_stride)
42:     MULT(A_4, B_4, C_8, new_n, log_stride)
43:
44:     for i in 0 .. (new_offset − 1) do
45:         for j in 0 .. (new_offset − 1) do
46:             C_1[j + i ∗ stride] += C_5[j + i ∗ stride]
47:             C_2[j + i ∗ stride] += C_6[j + i ∗ stride]
48:             C_3[j + i ∗ stride] += C_7[j + i ∗ stride]
49:             C_4[j + i ∗ stride] += C_8[j + i ∗ stride]
```

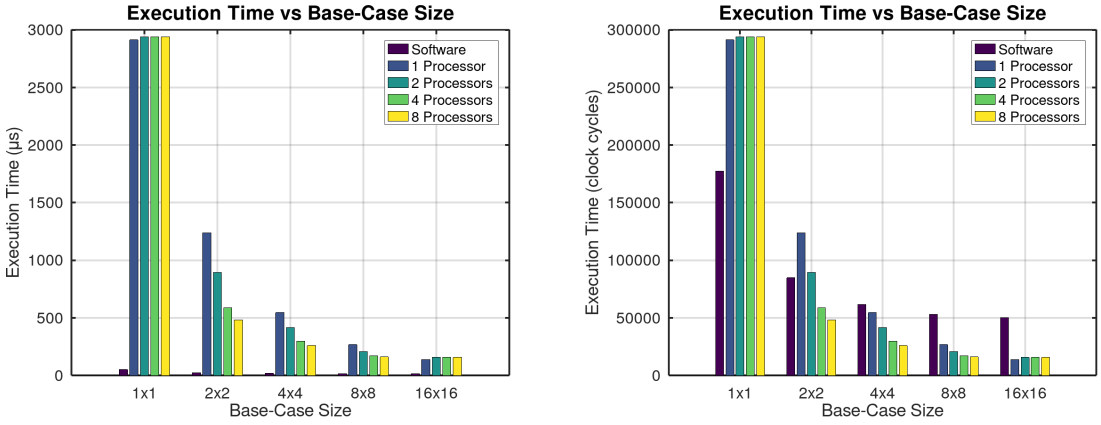
Table 5.10: Matrix Multiplication Energy Consumption (μJ)

Base Size	Software	1 Processor	2 Processors	4 Processors	8 Processors
1	1,999.28	1,313.36	1,449.14	1,552.02	1,978.24
2	929.37	572.40	442.04	325.67	329.20
4	661.74	243.97	199.28	155.59	169.17
8	564.39	120.14	100.67	95.45	112.31
16	529.75	61.21	73.60	85.94	106.25

energy in some cases.

The data plots in Figure 5.4a and Figure 5.5 make it clear that the most effective means of decreasing both the execution times and energy consumption is by increasing the size of the base-case. The hardware that was both the fastest and consumed the least power was when the base-case size was equal to the problem size. However, in this case the use of multiple processors is pointless since no function calls are generated other than the initial one.

The area usage for the block matrix multiplication implementation that uses a 4×4 matrix multiplication as its base-case is shown in Table 5.11. Again, we can use a linear regression to get an approximate cost for each additional processor, finding that each processor adds about 2,810 LUTs, 4,267 registers, and 1 BRAM. We can also



(a) Matrix multiplication execution times measured in μs.

(b) Matrix multiplication execution times measured in device clock cycles.

Figure 5.4: Matrix Multiplication Execution Times

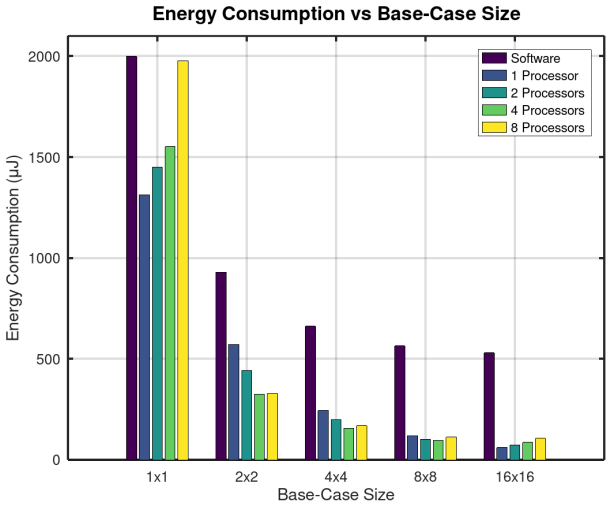


Figure 5.5: Matrix Multiplication Energy Consumption

estimate that the base system without any processors would consume 2,461 LUTs, 6,528 registers, and 101 BRAMs.

Note that the size of the block matrix multiplication system without any processors is significantly larger than the size of the previous two algorithms. One reason for this is that the matrix multiplication algorithm was tested with a much larger external memory size than quicksort, which accounts for significantly higher BRAM usage. Another reason is that since the matrix multiplication algorithm involves eight recursive function calls, all of its data structures must be significantly larger than the

Table 5.11: Matrix Multiplication Area Usage with 4×4 Base-Case Size

Resource	1 Processor	2 Processors	4 Processors	8 Processors
LUTs	4,998	8,079	14,183	24,737
Registers	10,699	15,140	23,648	40,633
Block RAMs	102	103	105	109

previous two algorithms, requiring more hardware resources.

Chapter 6

Conclusions

A new approach to implementing recursive algorithms in hardware, named TreeRecur, was explored with the goal of exploiting the procedure-level parallelism of divide-and-conquer algorithms. It was found that TreeRecur is a flexible framework that is capable of implementing a variety of different algorithms by only changing the front-end processors, which are defined with HLS. Furthermore, it was found that TreeRecur can achieve execution speeds comparable to software when the difference in clock speeds are accounted for, and had energy consumptions that were up to 11.2 times better than software.

For the hardware implementations, the best performances in terms of both execution speed and energy consumption occurred when the algorithms were tested with relatively large base-cases sizes. This is presumably a consequence of the higher function call overhead resulting from using a tree data structure instead of stack because larger base-cases mean that fewer function calls are made.

6.1 Future Work

There are many potential avenues for future work, including scaling the system to handle larger problem sizes and optimizing the back-end to reduce bottlenecking. There is also a lot of research that can be done regarding which kinds of algorithms are best-suited to be implemented with TreeRecur and how to optimize the HLS

code for the front-end processors in order to maximize performance. Finally, additional research could help improve TreeRecur's error handling. The system currently throws an error when it runs out of memory, but it might be possible to recover from such errors by dropping some function calls that are executing in parallel, since the sequential execution of recursive functions would consume less memory.

Bibliography

[1] Xilinx, *Vivado Design Suite User Guide: High-Level Synthesis*, Oct. 2019, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf.

[2] V. Sklyarov, "FPGA-based implementation of recursive algorithms," *Microprocessors and Microsystems*, vol. 28, no. 5-6, pp. 197–211, 2004.

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

[4] G. Ferizis, *Mapping Recursive Functions To Reconfigurable Hardware*, Ph.D. thesis, University of New South Wales, Australia, 2005.

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

[6] 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.

[7] G. Stitt and J. Villarreal, "Recursion flattening," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*. ACM, 2008, pp. 131–134.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, The MIT Press, 3rd edition, 2009.

[9] Xilinx, *7 Series FPGAs Memory Resources User Guide*, July 2019, https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf.

[10] A. Aho, M. Lam, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[11] S. Lahti, P. Sjövall, J. Vanne, and T. D. Hämäläinen, "Are we there yet? a study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 898–911, 2019.

[12] K. Rupnow, Yun Liang, Yinan Li, and Deming Chen, "A study of high-level synthesis: Promises and challenges," in *2011 9th IEEE International Conference on ASIC*, 2011, pp. 1102–1105.

[13] Intel, *Intel High Level Synthesis Compiler: User Guide*, June 2020, htpps://www.intel.com/content/www/us/en/programmable/documentation/ewa1457708982563.html.

[14] V. Sklyarov, 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.* IEEE, 2005, pp. 235–240.

[15] 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*, Dec 2010, pp. 121–126.

[16] *Processor Counter Monitor*, htpps://github.com/opcm/pcm.

[17] R. Dementiev T. Willhalm, *Intel Performance Counter Monitor - A Better Way to Measure CPU Utilization*, Intel, Jan. 2017, htpps://www.intel.com/software/pcm.