Composing Parallel Runtime Systems: A Case Study in How to Compose the Julia and OpenCilk Runtimes

by

Tim Kralj

B.S., Computer Science and Engineering, Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by .....
Charles Leiserson
Professor
Thesis Supervisor

Certified by .....
Tao B. Schardl
Research Scientist
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Composing Parallel Runtime Systems: A Case Study in How to Compose the Julia and OpenCilk Runtimes

by

Tim Kralj

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Julia [5] [15] is a high-level computing language used by many developers for its performance and ease of use. Julia operates on tasks that are run concurrently on threads. In its current state, however, Julia is not able to effectively employ fine-grained parallelism. OpenCilk [9] is an open-source implementation of the Cilk concurrency platform designed to utilize fine-grain parallelism. The Cilk runtime system, based on Cheetah [12], offers provably efficient parallel scheduling whose performance is borne out in theory and practice. I propose a combination of the Julia and OpenCilk runtimes through the integration of multiple components. One contribution of this thesis is a novel algorithm for combining C/C++ memory allocations with Julia's precise garbage collector. Composing the parallelism of OpenCilk and Julia enables programmers to write efficient multithreaded code. Additionally, this work is a case study of combining the high levels of parallelism present in Cilk with a high-level language.

Thesis Supervisor: Charles Leiserson
Title: Professor

Thesis Supervisor: Tao B. Schardl
Title: Research Scientist

Acknowledgments

I would like to thank various people for their contributions and guidance on this thesis: Valentin Churavy and Takafumi Arakaki for their guidance and help with Julia. They have more knowledge about Julia than I would ever hope to know. Special thanks should be given to my thesis advisors, Tao Schardl and Charles Leiserson, for their guidance, talks, vision, and continual support. Without TB this thesis would not be possible and his support this year was unrivaled.

Thank you to my family and friends who supported me through my years at MIT. Life here is fun and challenging but you all helped me through it.

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003965. This research was sponsored in part by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Contents

List of Figures

Chapter 1

Introduction

Julia is used as a high-level computing language built to be highly efficient and is used for many applications. In its current state, however, Julia is not able to effectively employ fine-grained parallelism. Julia supports asynchronous tasks which users create. The tasks are scheduled over several threads that execute the tasks concurrently. The tasks may run in parallel using shared memory to execute work. Julia creates as many threads as necessary to execute code concurrently. The concurrent threading model is beneficial and gives programmers the ability to write efficient parallel code; however the concurrent task model does not achieve the parallel performance Julia is so well known for in serial execution.

OpenCilk is a system designed for high levels of parallelism utilizing Cilk workers. The Cilk language is built upon *spawn* and *sync* functions to expose opportunities for parallel execution. The *spawn* function indicates the function which was spawned may, but is not required to, execute in parallel. The *sync* function creates a barrier that ensures all outstanding child tasks executing in parallel have completed before continuing. Cilk computations are scheduled and load-balanced using randomized work stealing on Cilk workers. The parallelism of a Cilk program can be measured in terms of two performance measures: work and span. The work, denoted by T_1, of a computation corresponds to the running time of the program on a single processor; the span, denoted by T_∞, of a computation corresponds to the running time of the program on a machine with infinite processors. The parallelism of a computation

```
function fib(N)
    if N <= 1
        return N
    end
    x = Ref{Int64}()
    Tapir.@sync begin
        Tapir.@spawn begin
            x[] = fib(N-2)
        end
        y = fib(N-1)
    end
    return x[] + y
end
```

Figure 1-1: Julia code of Fibonacci. The *Tapir.@sync* block specifies to the Cilk runtime the tasks in this section must all complete before continuing. The *@Tapir.spawn* signifies the execution of $X[] = fib(N-2)$ may, but not required, to happen in parallel to $y = fib(N-1)$. The tasks tagged with Tapir constructs are run on Cilk workers and scheduled by the Cilk scheduler using Cilk's work stealing algorithm.

is the ratio of its work divided by its span: T_1/T_∞. The Cilk scheduler provides a mathematical guarantee to execute a Cilk computation with work T_1, span T_∞ on P processors in time $T_p \leq T_1/P + O(T_\infty)$ [21]. If a Cilk program exhibits ample parallelism, $P << T_1/T_\infty$, then the Cilk scheduler executes the computation with almost linear speedup. This gives the user great flexibility to write efficient parallel code.

OpenCilk was developed as an extension to C/C++. This thesis proposes to create an integration of the Julia and OpenCilk runtimes to enable a programmer to be able to run parallel code in Julia as in Figure 1-1. The spawned code in Figure 1.1 is run in parallel on Cilk workers while inside of Julia. The workers are able to utilize all the parallel benefits of Cilk while running on the Julia runtime.

Two runtimes that behave in different ways and are built different do not easily blend. One of the major issues is memory allocation. Julia is a high-level language that incorporates a garbage collector. The garbage collector allows a user to not worry about allocating memory while offering greater flexibility. Other commonly used garbage collected languages are golang [2] and Java [4]. OpenCilk, on the other hand,

uses the classic parallel memory allocation scheme from C/C++. A user allocates space on the stack and the C language provides an internal allocator that the user interacts with to request memory. When memory is freed, the internal allocator maintains the memory for future use by the program. The memory models of Julia and OpenCilk are currently not compatible—a user cannot use the parallel workers of Cilk while assuming a garbage collected model at the same time. Julia does not know how to allocate objects on the stack for a user when operating on a Cilk worker.

This thesis explores how the OpenCilk runtime can be integrated into the Julia runtime. The goal of the integration is to execute Julia code on OpenCilk workers. Julia will create tasks and work to be executed while OpenCilk will perform the work in parallel as specified by the user using Cilk constructs. OpenCilk workers were created to operate on work efficiently in parallel and this thesis helps Julia understand how to execute work on OpenCilk's workers. The combination takes advantage of existing mechanisms in both Cilk and Julia requiring a small number of lines of code changed to existing mechanisms. The integration does not change how a user interacts with Julia while adding parallel constructs. The benefits of the integration are large: users can spawn parallel code in Julia with simple keywords such as *sync* and *spawn*. Cilk workers are very effective at work stealing [17] to provide large amounts of potential speedup in parallel programs.

My thesis explores a novel algorithm for integrating a garbage collected language with the cactus stack used by the OpenCilk runtime using Cilk hyperobjects. The design is made to change Julia and OpenCilk minimally and use constructs which already exist. The combination also includes an approach to startup of Cilk to work with Julia. Another goal of this work is to provide a case study in combining high-level languages with OpenCilk.

The rest of this thesis is as follows. Chapter 2 provides an overview of OpenCilk and Julia necessary for this thesis. Chapter 3 gives an overview of the design for integration. Chapter 4 provides an in-depth implementation explanation. Chapter 5 evaluates the integration. Chapter 6 discusses related work. Chapter 7 suggests future work for the integration.

Chapter 2

Background

This chapter covers requisite background knowledge for this thesis. Section 2.1 gives an overview of OpenCilk. Section 2.2 gives an overview of the Julia runtime.

2.1 OpenCilk Overview

Cilk [17] is a multithreaded language for parallel programming that generalizes the semantics of C and C++ by introducing linguistic constructs for parallel control. Cilk control constructs allow sections of the program to execute in parallel. The language uses a "work-first" principle to minimize the scheduling overhead borne by the work of a computation. The language uses just three keywords: $cilk\_spawn$, $cilk\_sync$, and $cilk\_for$. $cilk\_spawn$ indicates the spawned function may execute in parallel. The code immediately following the $cilk\_spawn$ is allowed to execute in parallel with the invoked child function. The keyword $cilk\_sync$ acts as a barrier that ensures that spawned child computations executing in parallel have completed before proceeding. The keyword $cilk\_for$ indicates iterations of a loop may happen in parallel. Figure 2-1 is an example of a simple program operating with Cilk. The function A is tagged with $cilk\_spawn$ to indicate it may execute in parallel. The $cilk\_sync$ afterwards creates a fence that waits upon work of the spawned children of the current function to complete before moving on .

```
void main () {
    cilk_spawn A ();
    B ();
    cilk_sync;
}
```

Figure 2-1: An example of a Cilk section. Function A is semantically labelled as parallel work that can be executed at the same time as function B. Once both A and B complete, the function can move past the sync and continue executing.

2.1.1 Cilk Runtime System

The Cilk runtime system [12] uses a provably-good work-stealing scheduler [16] to automatically distribute work among a set number of workers. Workers initialize at the start of the program and are built upon POSIX threads [10]. Each worker maintains a deque of tasks it needs to run. A worker W_1 may have work and interacts with the bottom of its own deque, treating it as a stack. This worker is executing but has extra tasks at the top of the deque which can be executed in parallel. If another idle worker W_2 wants to steal extra parallel work from W_1's deque, it steals from the top of the deque. W_2 is treated as a thief who is constantly looking at other workers to see if they can steal any work. The work stealing algorithm functions utilizing idle workers, called thieves, to search for continuations to steal. If a thief finds a continuation, it will steal the work and start executing.

The Cilk runtime maintains a collection of data structures in order to implement its work-stealing scheduler. It maintains a pool of fibers, blocks of memory that tasks use as stack space. The pool of fibers is created upon runtime startup allowing any worker to use the shared stack space. Whenever a task or continuation is stolen, the runtime allocates a new fiber for the function's exclusive use. These fibers are linked together to be able to backtrack through the fiber stack. The linking of fibers forms a cactus stack [23], a tree of stack allocated objects, inside of Cilk. Whenever a new fiber is allocated, a new block is added to the cactus stack with a parent pointer. The cactus stack is used to implement the runtime stack used by Cilk workers. Because Cilk workers steal continuations, newly stolen work needs to contain local state to

```
void B () {
    cilk_spawn D();
    E();
    cilk_sync;
}
void A () {
    cilk_spawn B();
    C();
    cilk_sync;
}
```

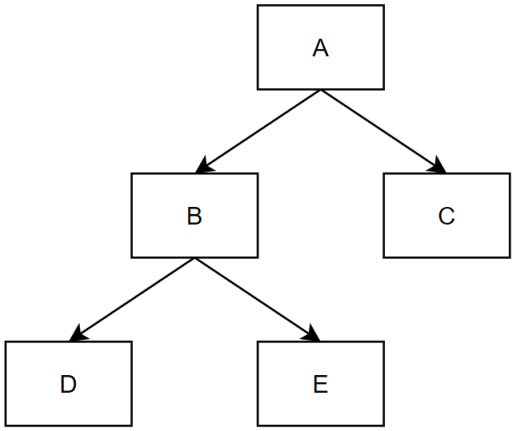

Figure 2-2: A simple Cilk program with its corresponding cactus stack. The Cilk program creates 5 functions to execute with some in parallel. The figure on the right is the cactus stack created from the Cilk program execution.

the computation. The cactus stack provides the structure for workers to dynamically switch work and maintain state of computations. The nodes are removed when the relevant computation completes and are popped off the tree.

The runtime also maintains Cilk stack frames. Whenever a Cilk program enters a spawning function or a spawn helper, it creates a Cilk stack frame, analogous to creating frames for normal function calls. The Cilk stack frames are used to store continuations that can be stolen and maintain state for each spawning function and spawn helper. These stack frames are then passed to the runtime calls behind *cilk_spawn* and *cilk_sync*, in order to properly maintain Cilk's internal stack state, called the cactus stack. When a worker steals work and the runtime allocates a new Cilk stack frame, the stack frame is added to the cactus stack. The cactus stack maintains a chain of Cilk stack frames for which the workers executed. At the end of the function, the runtime exits the frame and removes it from the cactus stack.

Because Cilk stack frames are created for spawning functions or spawn helpers, chaining together Cilk stack frames gives us a global view of a Cilk program's stack state. Figure 2-2 illustrates what the cactus stack for a simple Cilk program, on the left of the figure, would look like. The Cilk stack frame for a spawner function

```
int fib (int n)
{
    if (n<2) return n;
    else {
        int x, y;
        x = cilk_spawn fib (n-1);
        y = fib (n-2);
        cilk_sync;
        return (x+y);
    }
}
int main (int argc, char *argv[])
{
    int n, result;
    n = atoi(argv[1]);
    result = fib(n);
    return 0;
}
```

Figure 2-3: Fibonacci written in C using Cilk keywords. $fib(n-1)$ may execute in parallel on a different worker as $fib(n-2)$. No thread may execute code after the *cilk_sync* until the work of child computations in the spawning functions have finished.

(like function A or B) can have multiple child frames, but frames can only have one parent. Each task's local view of the cactus stack corresponds with its call stack. The cactus stack is not equivalent to a C/C++ stack and is not present in Julia's stack allocations. Let's consider an example with a single worker, W_1, executing the code in Figure 2-2. This worker executes section blocks in order of: A, B, D, E, C. The worker will have a stack view at any point of the Cilk stack. During execution of block D, the view the worker has of the stack is A -> B -> D–similar to a C/C++ view of the stack. Let us now have 2 workers, W_1 and W_2, executing in the code block. While W_1 is executing block D, W_2 may be executing block E because of a steal of the continuation. In this case the workers would have different views of the stack but share the exact same B and A blocks. W_1's view of the stack would be the same as before while W_2's view would be A -> B -> E. Each worker could have a different view of the stack, while sharing some elements, forming the cactus stack structure.

The Cilk runtime uses Closures to manage the coordination of the stack view of a worker. In the runtime, tasks are represented as Closures (also called full frames), structs that contain the fiber and Cilk stack frame they are currently executing. These closures allow workers to switch work execution and pick up other work execution. On a steal, a worker picks up the task and local state from a Closure. A join counter, which represents the number of outstanding children (i.e. the number of children we need to wait for before we can proceed past a *cilk_sync*) is needed to track how many outstanding jobs must occur before the sync may be executed. The entries of a worker's deque are pointers to Cilk stack frames. There is 1 Closure associated with a workers deque at any time, and it is associated with the top most stack frame of the deque. Closures also contain pointers to their parent closure, their left sibling closure, and the right sibling closure. Workers maintain a busy leaves property: a closure tree exists only if some leaf has a worker executing work. Closures may be suspended by any workers when children are executing spawned code.

2.1.2 Cilk Reducer Hyperobjects

Cilk supports reducer hyperobjects, which allows concurrent updates to a shared variable or data structure to occur simultaneously without contention [20]. The reducer acts as a variable that can be shared safely between many threads. A reducer is defined in terms of a binary associative REDUCE operator. The operator can be a sum, list concatenation, etc. Updates to the hyperobject are accumulated in local views, which the Cilk runtime system combines automatically when subcomputations join. When a reducer is active, each worker has their own view of the nonlocal variable. Each view is tied to the Closure in execution to allow worker stealing to occur separately. On a steal, the worker updates the local view of a reducer to match that from the stolen Closure. As we shall see, the Cilk-Julia integration uses a reducer to store garbage collection state per worker for Julia during execution.

Reducers operate in a way that allows workers to have local views of the same variable when executing parallel strands and then internal mechanisms "reduce" the views together when the work completes. For example, a summation reducer, like

```
// creating reducer:
typedef CILK_C_DECLARE_REDUCER(int) ExampleReducer;

ExampleReducer n = CILK_C_INIT_REDUCER(int,
  sum_reduce, sum_identity, sum_destroy, 0);

void sum_reduce(void* key, void* left, void* right) {
    *(int*)left = *(int*)left + *(int*)right;
}

void sum_identity(void * key, void * value) {
    *(int*)value = 0;
}

void sum_destroy(void* key, void* value) {
}

void sum_function(int i) {
     // accessing Cilk reducer on a worker:
    REDUCER_VIEW(n) += 1;
}

void main () {
    cilk_for (int i = 0; i < 1000; i++) {
        sum_function(i);
    }
    int sum = REDUCER_VIEW(n);
}
```

Figure 2-4: The initialization, identity, reduce, destroy, and access pattern of a reducer. Cilk workers can execute $sum\_function()$ in parallel followed by retrieving summed value in sum.

in Figure 2-4, may have multiple local views of the variable n, during the reduction operation occurs all the views are summed together to get the final sum. This feature allows for a thread-safe model of many different views of a variable maintained by the runtime.

A reducer requires 3 functions associated with it. These are: the identity function, the destroy function, and the reduce function. The identity is used to create a new view of the reducer variable. This can occur on a new Closure or the first access by a worker. The destroy is used to delete a view on completion of the local view.

The reduction is used when merging multiple views together. This can occur when a subcomputation completes. The sum example mentioned earlier shows how to reduce the hyperobject. Figure 2-4 illustrates the identity, destroy, and reduce functions for the reducer n as well as the creation and access pattern of the reducer on Cilk workers.

A reducer is a powerful construct built into the Cilk language and is useful for having different tasks maintain local variables.

2.2 Julia Overview

Julia is a high-level language made for general applications with a focus on numerical analysis and computational science. Julia was made to process mathematical models and has expanded to general computing. The language is written in C/C++ and compiled down to LLVM [6], a compiler intermediate representation (IR). Julia has many features of other high-level programming languages such as garbage collection. This section will be an overview of Julia's runtime relevant to the Cilk-Julia integration.

Julia maintains a runtime that executes code written in the Julia language. The components are tasks operated on in a threading model. A simple example of a Julia program performing binary search is in Figure 2-5.

2.2.1 Threading Model

Julia's threading model contains concurrent threads used to execute work. The threads are built on libuv[11] using POSIX threads[10], same as Cilk. The Julia system creates tasks, sections of work execution, that execute on Julia threads. Tasks are used in Julia as a control-flow feature that allows computations to be suspended and resumed. A Julia programmer writes a program which utilizes tasks. Syntactically, a user of Julia can specify an aspect of a program to run in parallel using @*spawn* [8] syntax before an expression. The expression following the macro will run on an automatically-chosen process by the runtime. The Julia runtime can then choose to execute that program on system threads. Concurrent tasks become useful for exe-

```
function binarySearch()
    u = T(1)
    lo = lo - u
    hi = hi + u
    @inbounds while lo < hi - u
        m = midpoint(lo, hi)
        if lt(o, v[m], x)
            lo = m
        else
            hi = m
        end
    end
    return hi
end
```

Figure 2-5: An example of a Julia program performing binary search.

cution that benefits from having threads that rely on each other for execution. The threads are concurrent to allow the CPU to switch between different threads during execution of a program.

2.2.2 ptls Pointers

Julia maintains threads that operate together. The runtime tracks information about each of these threads inside of their respective thread local storage (TLS) [19]. The runtime initializes important components about each thread as the *ptls* variable that resides on the thread itself. The *ptls* enables the runtime to track many components about each thread and the task executing on the thread. The *ptls* enables the runtime to store and lookup important state about each thread in the thread itself instead of a global variable in the runtime. The thread-local state is very important in Julia. It keeps track of tasks, thread states, garbage collection states, and more.

2.2.3 pgcstack

One important part of the *ptls* in each thread is the *pgcstack*. The *ptls* component of the thread tracks all the stack allocations that occur. The *pgcstack* is the head of a linked list of all roots a thread has created. The roots are created on the stack

when memory is allocated by the program. This linked list grows larger whenever more roots are made on this thread and roots of the linked list are removed upon popping of the stack. The linked list can be used to identify any roots which have been created on this thread during execution and free the sections of the stack which are no longer useful for the program execution–the garbage.

The *pgcstack* allows Julia's garbage collector to be precise. Precise garbage collectors can correctly identify all pointers in an object. The opposite would be conservative garbage collection which assumes that any bit pattern in memory could be a pointer if, interpreted as a pointer, it would point into an allocated object. Julia's garbage collector mandates a stack allocated linked-list, *pgcstack*, on which all heap references currently in use are rooted.

2.2.4 Garbage Collection

Julia utilizes a garbage collector. A user does not malloc or free memory such as in the C/C++ model, instead memory is allocated and reused without the user's input. The Julia garbage collector is mark-sweep/tracing, non-compacting, precise, and generational. The runtime works as normal executing user code and creating roots whenever necessary. When we reach a threshold for the amount of garbage in our system, garbage collection initiates.

The collector is lead by a single thread and iterates over all the roots, pointers to objects on the heap, to discover which ones are necessary for the program going forward while the unused are swept away. In the mark phase the garbage collector starts from *pgcstacks* and walks through the object graph marking objects that are reachable. If objects are necessary for future execution they exist as roots inside a *pgcstack* and are marked. The *pgcstack* mechanism is illustrated in Figure 2-6. Every root in the scope of workers j_1 to j_4 can be found by walking the linked list of nodes starting at A, B, C, and D. These are the nodes contained in the *ptls* of each Julia thread. After the mark phase, the garbage collector sweeps away all objects that are not marked, thereby reclaiming the memory. The garbage collector is non-compacting since it does not move any objects during the mark or sweep phases, and the garbage

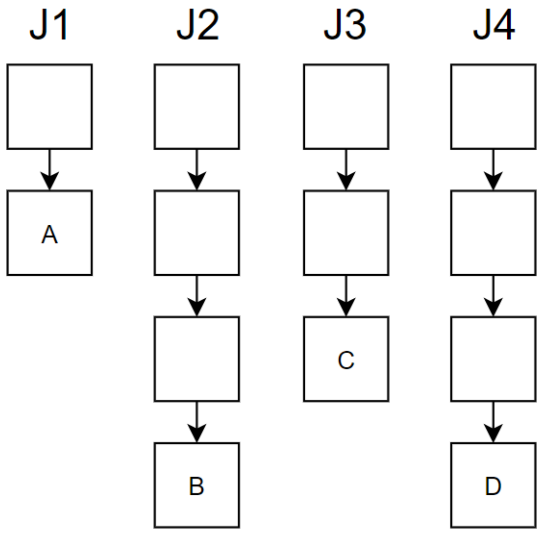

Figure 2-6: The *pgcstack* scanning mechanism. Every column is the stack for a J-thread J_1 to J_4. Each node represents roots on the stack with a previous pointer. On garbage collection, every root in the system can be discovered by unwinding the linked lists starting at A, B, C, and D.

collector forms a generational map for doing full scans or new scans. Julia's garbage collector operates on the generation hypothesis–the observation that, in most cases, young objects are much more likely to die than old objects. Old objects that have stayed in the program for a while are most likely to stay in the program for longer. Julia takes advantage of this and has an option to do smaller garbage collections on newer objects only.

Any thread can start garbage collection and continues to lead garbage collection until the end of garbage collection. This thread will scan every root including its own roots. In addition, threads must all come to safepoints before garbage collection moves to the mark and sweep phases. Threads safepoint by coming to a stop during execution and waiting until the garbage collection has ended to continue. Stop the world execution is necessary to not create any new roots during garbage collection. Each Julia thread will find a spot in their execution where it is determined safe to stop. The leading thread iterates through the linked list of roots in the *pgcstack* values stored in the *ptls* mentioned above. All of the roots can be found by following

each linked list to the head, every object not reachable from one of these roots will be swept away.

Garbage collection has many steps that are important to the Cilk-Julia combination. The items that occur are listed in order below:

1. Garbage created on the heap (normal operation)
2. Garbage collection initiated
3. Each thread comes to a safepoint
4. (mark) Garbage collector scans through thread stacks to find roots
5. (sweep) Reclaim space occupied by unmarked objects
6. Threads resume as before

2.2.5 LLVM

Julia lowers code to LLVM [6] during compilation. LLVM is a set of compiler technologies used to develop programming languages. LLVM takes intermediate representation (IR) code from a compiler and emits optimized IR. Julia uses the compiler to produce fast code to run on different systems. Cilk also uses LLVM with an extension containing parallel constructs in the Tapir compiler [25]. The combination of languages on the compiler aspect aids in adding parallel constructs in Julia which Cilk also understands. To add parallel constructs in LLVM, Julia can switch the backend ABI used to the Tapir/LLVM compiler.

Chapter 3

Runtime Integration

The Julia and Cilk runtimes were created in different ways and need to learn to interact together. The goal of this thesis is to provide a method for combining the two and a case study for future runtime integration's. Section 3.1 describes how Cilk workers deal with stop the world execution in Julia. Section 3.2 describes how Julia handles the parallel cactus stack present in Cilk and garbage collection on Cilk workers. These parts all orchestrate together to enable the integration of Cilk and Julia.

There are other approaches that may work for the combination of two runtimes. Chapter 6 focuses on previous work for the combination of threading models, and Chapter 7 focuses on future work.

3.1 Stop the World in Cilk

As described in section 2, Julia garbage collection requires stop the world execution. All the julia threads in the runtime come to a stop each time garbage collection is performed. Cilk workers must do the same to allow Julia to garbage collect on top of each Cilk worker. This section will describe the algorithm for stop the world execution in Cilk.

Cilk workers are created on top of pthreads [10]. These threads are made to execute a specified function until they are destroyed, and Cilk maintains a work

stealing loop until the runtime shuts down. The work stealing loop has 2 options for a worker: executing work or searching for work to seal. The workers are a thief when they search for work and steal some when they find it.

The integration takes advantage of this continuous loop of searching for work. The Julia runtime sends stall work tasks to Cilk workers similarly to the mechanism used for waiting on Julia threads in the Julia runtime. During a safepoint, threads sleep constantly until garbage collection has ended. The Cilk runtime does not know what is going on outside of its own scope and Julia sending instructions that allow the current workers (not thieves) to loop continuously while waiting acts as a stall. Such a mechanism allows the workers to be in a "stop the world" scenario like Julia requires for garbage collection. The purpose of the halt in execution is to allow the garbage collector to run through the pointers of the program to mark and sweep them. No garbage should be made during the mark or sweep phases to not race on new pointers being created. The Cilk runtime integration of sending a stall loop achieves this goal and allows for garbage collection to occur. Thieves do not have any relevant roots for garbage collection and are always in a safepoint state until they start executing Julia work.

3.2 Garbage Collection Algorithm

This section includes a novel algorithm for integrating a garbage collector based language with the cactus stack used by the OpenCilk runtime using Cilk hyperobjects. The Julia runtime creates roots on Cilk workers and requires garbage collection to occur on the workers to remove the generated garbage, as described in section 2.

3.2.1 Design Considerations

This section will describe the possible routes for the OpenCilk and Julia integration. As described in chapter 2.2, the Julia runtime requires some knowledge about the state of Cilk workers to perform garbage collection. Each worker will perform allocations on the cactus stack to generate roots. These roots are tracked in Julia using thread-local

storage.

An obvious first route for the integration would be to track the same state inside of Cilk workers TLS. Each worker can maintain their linked list of allocations using a *pgcstack* inside of their respective TLS. The garbage collector can discover all roots which exist on each thread. An example can be used from Figure 2-2: one can imagine 3 workers in operation. Worker 1 executes blocks A, B, and D; worker 2 executes block E, and worker 3 executes block C. During a steal of new work or stack allocation, a worker updates their thread-local *pgcstack* variable to append the new roots. The linked list of allocations would correspond to the view of the Cilk cactus stack of each worker. Worker 1's linked list would contain allocations from A, B, and D, worker 2's contain A, B, and E, and worker 3's A and C.

One initial fault with this design is double counting roots. The roots contained in block A would be discovered in all 3 worker linked list states. Double counting is acceptable for the garbage collector, but the algorithm would not be efficient if workers share allocations higher in the stack. Some form of exclusion algorithm could be created to only count objects once but the duplication of objects still exist on the TLS of workers. Tracking TLS states during randomized work stealing also requires new logic in the OpenCilk runtime. When a worker becomes a thief followed by stealing new work, a new fiber is allocated and a new *pgcstack* is added to the linked list. This is great for the tracking of new allocations and executing a continuation; however, new complexity is added with parent pointers to states. These pointers would need to be tracked across longjumps [1] in the code which is not an easy task. One question that arises is what occurs when execution of block B has completed and the roots are no longer necessary for the program (now garbage) on the linked list of worker 2? Each worker containing block B in their linked list of stacks in TLS should remove the node from their linked list. This is not an easy task and asking workers to shift away from work on the program is against the principles of OpenCilk.

All of these drawbacks of using TLS in a similar manner to Julia threads lead to a different design which is simpler, more efficient, and easier to implement. The design will be described in the following sections.

3.2.2 Finding Roots on Cilk workers

The Julia runtime creates garbage during/after tasks are completed. Roots are necessary for a program to have local state for operation; however, once state is no longer useful it becomes garbage. The current mechanism Julia uses to create roots on the stack can be re-purposed to create roots on Cilk workers. As mentioned in section 2-2, each thread inside of Julia has a pointer to the head of the linked list of roots, the *pgcstack*, inside of their thread local storage. Cilk workers also have thread local storage, but the work stealing algorithm of Cilk workers means the workers could be changing the space they are operating on at any time. In the subsequent sections, the mechanism for maintaining a correct *pgstack* is described. Whenever Julia needs to create roots on a stack it uses the same mechanism for Julia threads and Cilk workers. The Julia runtime retrieves, the algorithm described below, the current *pgcstack* of a worker, creates a new node containing the roots and updates the head of the linked list with a previous pointer to the previous *pgcstack*. The depiction of the system is shown in Figure 2-6. This mechanism allows the Cilk stacks to grow with new roots as long as the correct *pgcstacks* are managed. The following section describes the algorithm to do so.

3.2.3 Cilk Garbage Collection Algorithm

Julia can sends Cilk workers tasks to execute through the LLVM compiler integration when parallel constructs in Julia mark functions such as in Figure 1-1. These tasks will generate roots and garbage as mentioned in the previous section. Each worker must maintain a *pgcstack* containing all the roots in the program for garbage collection to occur correctly. The *pgcstack* cannot be stored in the same way in a Cilk worker as a Julia thread: workers perform work stealing and the stack linked list will not operate correctly with steals and syncs without large changes to tracking TLS inside of OpenCilk. The algorithm devised to fix this uses a central hyperobject to store the most current *pgcstack* for every worker. The workers can track their current *pgcstack* inside of the reducer, update it with new allocations during work, and work

steal when necessary while storing the correct state. The mechanism provides the ability to track roots created during current execution of work as well as any stolen work. The central hyperobject can save the state of the worker inside a Closure on a steal. This Closure, when restored by a worker on a continuation, will contain the previous value of the *pgcstack*. The reducer values are stored inside the Closure and workers can restore state from a Closure.

During the start of garbage collection, the leading garbage collector thread sends a safepoint instruction for all workers and Julia threads. Each worker which is currently executing work (not a thief) will come to a stop and wait as described in 3.1. The lead garbage collecting thread can now continue into the mark phase. During marking, the leading thread discovers all possible stacks containing roots with the union of 3 parts: Julia threads *pgcstacks* inside the *ptls* (as happens currently in Julia), the *pgcstack* inside each workers local view of the hyperobject, and any *pgcstack* inside of a suspended Closures. The combination of the 3 parts includes all roots in the system and are marked as such. The garbage collector thread continues to the sweep phase and removes any non-marked roots and execution continues on all threads/workers as normal.

The algorithm adds some complexity to the original Julia garbage collector; however, it reuses many components to remain efficient. The steps laid out in order:

1. Garbage created on the heap (normal operation) on Cilk workers and Julia threads
2. Garbage collection initiated
3. Each thread/worker comes to a safepoint
4. Cilk Root Discovery
5. (mark) Garbage collector scans through thread stacks to find roots
6. (sweep) Reclaim space occupied by unmarked objects
7. Threads resume as before

In subsequent sections will be a discussion on the hyperobject required to maintain stack information on Cilk workers using a hyperobject.

3.2.4 Cilk pgcstack Reducer

The idea of a reducer [20] is a hyperobject implemented in the OpenCilk runtime as described in section 2.1.2 The object lives as a central datastore and each worker can have their own local instance of an object, called a "view". For a single variable, x, each worker can have their own version of the variable to access and alter. Updates to the hyperobject are accumulated in local views, which the Cilk runtime system combines automatically when subcomputations join.

The integration can combine the ideas of a reducer and the *pgcstack* to maintain our cilk stacks. A central reducer will be used for all the *pgcstack* management throughout the usage of Cilk in Julia. The reducer hyperobject has a mechanism for every Cilk worker to maintain their *pgcstack* like a Julia thread does inside of their *ptls*. This reducer allows for the Cilk worker to access the current head of the *pgcstack* whenever roots are being created on the stack and allows the program to perform stack scanning during garbage collection.

3.3 In-depth Reducer and Closure Mechanism

In this section I will be going in depth into the reducer and Closure system. The reducer is used to maintain the relevant data needed to perform garbage collection on Cilk stacks.

First, a mathematical description of the reducer. For a given Cilk program, let h be a reducer with an associative operator $\otimes$. Let $G = (V, E)$ denote the computation dag (directed acyclic graph) modeling the Cilk program's execution, and let u and v denote two strands in V. The peers of u is the set of strands in the Cilk computation G that are logically in parallel with u. Consider a serial walk of G, and let a_1, a_2,..., a_k denote the updates to h after the start of strand u and before the start of strand v. Let $h(u)$ and $h(v)$ denote the views of h at strands u and v, respectively. If peers(u)

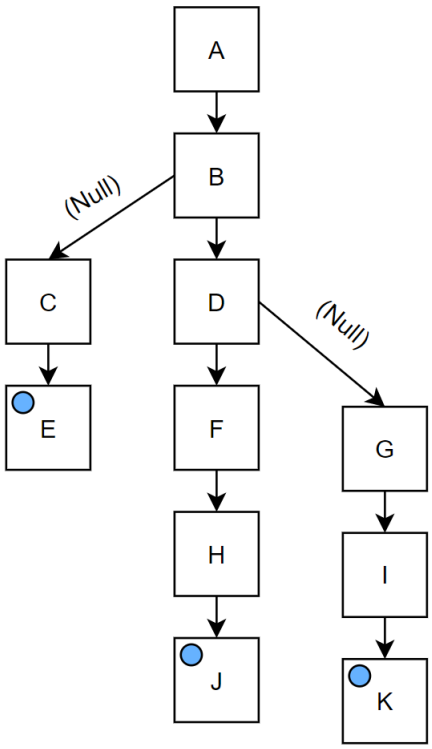

Figure 3-1: The linked lists of the *pgcstack*. Each box is a node on the linked list and contains roots in our program. On garbage collection, all roots can be found by scanning from E, J, and K up the linked list. There are no nodes which will be scanned twice. A blue node represents a *pgcstack* stored in the reducer.

$=$ peers(v), then $h(v) = h(u) \otimes a_1 \otimes a_2 \otimes ... \otimes a_k$.

3.3.1 Cilk stacks

Cilk performs work-stealing between different workers. This means at any point a piece of work could be stolen by a worker and the worker start allocating more variables on the stack–something the runtime integration must track. The reducer mechanism helps us with this: each worker has their own view and the view is changed whenever a worker changes the work they are operating on. The feature allows the Cilk runtime to halt at any point and scan through all the reducer views to find the relevant *pgcstacks*.

In this section I will reason about why we will have all the correct stacks even with

```
void pgc_stack_identity(void* key, void* value) {
    *(jl_gcframe_t**)value = NULL;
}

void pgc_stack_destroy(void* key, void* value) {

}

void pgc_stack_reduce(void* key, void* left, void* right) {
    assert(*(jl_gcframe_t**)right == NULL);
}
```

Figure 3-2: The reducer mechanisms for the central *pgcstack* reducer.

dynamic worker stealing and not miss any active roots during garbage collection. If we have a single worker iterating through work then the worker will have the correct view always on their *pgcstack*. When multiple workers are in play there is extra complexity. Every worker starts out as a thief with an empty *pgcstack*. No roots have been created yet. On a steal, the worker's view is created with the identity function to an empty *pgcstack*, without any roots, since this worker has not allocated any new objects if picking up a continuation from a different worker. The previous allocations of the continuation will be counted as roots in the *pgcstack* of the worker that created the objects. When a new allocation occurs, the stack will be updated to maintain it. When a worker pops out of a function on completion all the roots which are no longer necessary become garbage. This is represented in the hyperobjects by removing the strand view of the reducer. On a suspended Closure, the reducer view is saved inside the Closure itself. The mechanism allows the worker to switch the work they are executing while storing our roots inside a scannable field by the garbage collector: the Closure.

An example of the data-structure is shown in Figure 3-1. The linked lists are the stacks needed to be tracked. The execution of this program had 2 spawns occur at C and at G. When a worker picked up the continuation, their view of the reducer was an empty *pgcstack* as shown in Figure 3-2. At the start of garbage collection, the current reducer views for different workers are represented with a blue dot each. To

iterate over all nodes in the tree A->K, there are 3 necessary nodes to be found: E, J, and K. From these, the whole tree can be discovered with linked list unwinding. If there existed any suspended Closures, the reducer view of the *pgcstack* inside the Closure would contain the roots from the computation. These 3 nodes all exist within current reducer views of workers or suspended Closures.

A worker may pop up the stack when work is completed. Popping indicates the roots that were used in this section of work are no longer necessary and the worker can reset their *pgcstack* view to be the "left" branch before a potential steal occurred. The left branch of the reducer is one that contains all the previous *pgcstacks* in the linked list of prior computations. The right should be empty on a pop because the initial view of the reducer was empty. The worker asserts the right node is $NULL$ (indicating a sync and popping up the stack). The reducer view property maintains the idea that the left most branch was the previous stack pointer before the steal and any roots which are necessary in a program are maintained in the left branch.

Chapter 4

Implementation

As described in Chapter 3, there are many aspects that come together to form the OpenCilk and Julia runtime integration. There are 3 main areas of changes for the implementation: the Julia runtime changes, Cilk runtime changes, and the combination glue code. One of the goals of this thesis is to create a novel way to combine runtimes with OpenCilk that required minimal changes to either original runtime. The glue code, therefore, contains most of the changes and integration pieces.

This chapter will give a look at the implementation changes to both runtimes and the glue code required to make the integration successful. Section 4.2 will focus on the changes to startup and shutdown code. Section 4.3 will dive into the garbage collection alterations necessary.

There are many possible ways to combine two runtimes. This approach is a minimal extension design designed to alter both runtimes as little as possible. Chapter 5 will describe if the combination achieves the goals set out for it while Chapter 6 will examine at previous approaches at combining runtimes.

4.1 Design Decisions

An initial route the exploration of runtimes took was to rewrite parts of the OpenCilk runtime inside of Julia. Julia already has threads that run concurrent tasks, and users can create tasks within their code. The combination would entail porting much of the

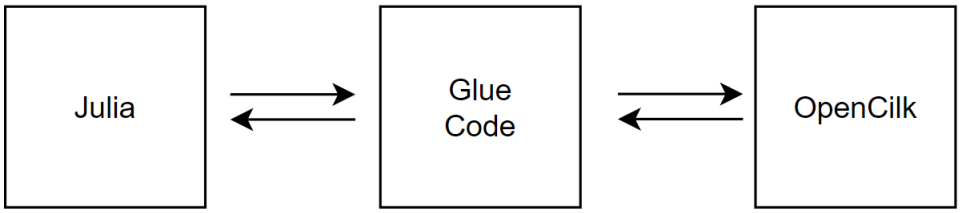

Figure 4-1: The design choice of integrating Julia and OpenCilk. Each runtime remains separate and a small layer of glue code orchestrates the two to work together. Julia can send Cilk workers tasks to execute and perform garbage collection whenever necessary.

OpenCilk runtime into Julia. Managing workers for randomized work stealing requires code to orchestrate them and Julia would be required to do so. There were some downsides when considering this approach. OpenCilk has many features bundled in the concurrency platform such as a scalability analyzer and a race detector [27] [24]. These additions are helpful for writing Cilk code and debugging but less useful for regular Julia threads and serial code. The extra aspects would not help users unless they were writing parallel code. Instead of going down the path of full integration, a design choice was made to have the runtimes of OpenCilk and Julia remain separate. A user can decide to use OpenCilk workers and runtime with Julia but they are not required to. Similar to Cilk's composition with C/C++, Cilk is a system which Julia can leverage for fine-grain parallelism but remains independent. The design in Figure 4-1 shows the integration with a small amount of glue code managing the combination. A benefit of the minimal extension design is neither code base of OpenCilk or Julia are altered in a substantial way. The runtimes stay separate but learn to interact with each other whenever necessary. Some goals for the design are simplicity and efficiency. The glue code contains a majority of the changes in this thesis.

4.2 Initializing Cilk in Julia

This section will cover the changes required to make Cilk initialize correctly in order to work with Julia and shutdown after the completion of work.

The Julia runtime starts at the initialization of a Julia program. In the startup steps, each of the Julia threads are set up and the components necessary to manage the runtime are started. Cilk also requires initialization functions to start workers and the runtime. In addition, Cilk has mechanisms to register callback functions. For the integration, Cilk Workers must behave close to the behavior of Julia threads. Initialization functions used to startup parts of the Julia threads are run on Cilk workers through a callback function. The callback function creates the *ptls* variable on the worker's TLS. Julia requires the *ptls* exists on any thread it interacts with, including Cilk threads. When the Julia runtime starts up, the system adds setup callbacks to the Cilk runtime. This allows each worker to initialize their *ptls*.

The initialization of the *ptls* on Cilk workers gives Julia some knowledge about the Cilk runtime. Julia is able to query the workers in the same way as it normally would with Julia threads. There are many aspects of the Julia runtime which rely on thread local storage appearing in a certain format and the initialization of *ptls* solves many of these. This includes the garbage collection state per worker, the $world-age$ of computations, and task information.

The central reducer is also created on the startup of the Julia runtime. The reducer is nested inside of the glue code and operates until the shutdown of the Julia runtime. The reducer allows Cilk workers and the Julia runtime to access *pgcstacks* on Cilk workers.

4.3 Cilk Garbage Collection

In this section I will describe the method by which Julia performs garbage collection with Cilk workers. The algorithm for garbage collection was described in section 2.2 and the updates for the Cilk integration in section 3.2. The steps of the Julia garbage

collector that occur in order are:

1. Garbage created on the heap
2. Garbage collection initiated
3. Each cilk worker comes to a safepoint
4. Cilk Root Discovery
5. (mark) Garbage collector scans through thread stacks to find roots
6. (sweep) Reclaim space occupied by unmarked objects
7. Threads resume as before

Each component will be described in a separate subsection below. These will cover the changes necessary to combine the Julia and Cilk runtimes.

4.3.1 Garbage Created on the Heap

Julia sends work to the Cilk workers and executes work for the Julia runtime. Functions are run on Cilk using parallel workers for work stealing as outlined in the Cilk [21] paper. Roots are created during normal worker operation inside of *pgcstacks* in the reducer. When work for a stack section is complete, the corresponding *pgcstacks* are popped and the allocated memory remaining becomes garbage. The unused memory needs to be reclaimed by the garbage collector.

One important component that differs is the accessing of variables on the *pgcstack*. Whenever more stack is allocated on Cilk workers, the access for *pgcstack* needs to be different from regular Julia accessing. These linked lists can be scanned during garbage collection to identify any roots. The Cilk workers have a different access pattern because *pgcstacks* live in the reducer. Whenever a worker steals work the current current place of the linked list head changes. As described in section 3.2.3, Cilk workers maintain a view of their current *pgcstack* in the reducer. On a new stack allocation or stack pop, Julia queries whether the code is executing on a Cilk worker

or Julia thread through an insertion call to LLVM code. The combination LLVM code handles each incoming request to a *pgcstack*. In the latter case the lookup is just as before: finding the previous *pgcstack* on that threads *ptls*. In the Cilk case, the Julia runtime identifies the *pgcstack* from the reducer on the executing worker. The current view of the stack for the worker is always up to date within the hyperobject. The lookup allows regular Julia threads to work normally while capturing all the roots inside of the *pgcstack* mechanism for a Cilk worker.

4.3.2 Garbage Collection Initiated

At some points during collection, the Julia runtime decides enough garbage has been created and one thread initiates garbage collection. The collection can occur on a Julia thread or a Cilk worker. The worker/Julia thread which initiated the garbage collector becomes the thread which leads garbage collection for the remainder of this garbage collection cycle. Garbage collection can occur on any worker so Cilk workers must have the ability to perform garbage collection. Julia's current mechanisms are sufficient in allowing a Cilk worker to run the garbage collector code.

4.3.3 Workers Find Safepoints

Each thread, Cilk worker or Julia thread, discovers a safepoint to stop at when trying to allocate more objects. As discussed in section 3.1, Cilk workers now have the ability to enter a wait loop. Julia sends instructions to Cilk workers whenever they try to allocate memory on the heap to safepoint. They stay in the looping phase until the lead thread/worker has completed the process of garbage collection.

Workers are not all executing Julia code. The Cilk worker thieves are executing the work stealing loop in the Cilk runtime and cannot reach a safepoint. They will never reach a safepoint and Julia would never continue executing. To account for this, the integration marks each worker that becomes a thief as being at a safepoint with a callback to Julia. On a steal, and therefore switching to executing Julia code again, the worker exits the safepoint section and moves back to regular execution,

with another callback, that is not safe to garbage collect upon.

4.3.4 Cilk Root Discovery

The garbage collector needs to find all roots in the runtime. In Julia this is done with *pgcstacks*. As mentioned above, these stacks form a linked list of roots. New roots create an extension of the *pgcstack* while popping roots removes them from the linked list. The garbage collector from Julia needs to know all possible roots in the Cilk runtime.To achieve this, two different parts of the Cilk runtime are scanned. The first of these are the current *pgcstacks* that exist in the reducer view of each worker. These are the current heads of the linked list which contains all prior roots created in this fiber (a section of work done on a worker) within this Cilk worker. All of the *pgcstacks* from the Cilk workers are combined into a list that can be scanned to discover all previous roots.

Another important area which could contain roots are suspended Closures, defined in section 2.1. Whenever a worker completes a section of work and becomes a thief, there is a possibility a child computation is still executing on a different worker. The steal caused the worker executing the continuation to have an empty *pgcstack* (creation of a new identity *pgcstack*, Figure 3-2) and therefore the roots in the suspended Closure are not captured in the *pgcstacks* of the Cilk workers. These are now dangling roots that also need to be marked. The Closure that contains work when the worker becomes a thief remains with roots that are vital to execution. The Closure becomes suspended and awaits for all child computations to complete. The suspended closures inside of the Cilk runtime need to be discovered for its root pointers. The suspended Closures are iterated through from the root of the Closure tree. Each Closure has its own local view of the Reducer view and allows the glue code to discover all remaining *pgcstacks* from suspended Closures in the Cilk runtime.

Together the list of *pgcstacks* in each of the workers and the *pgcstacks* in the suspended Closures is returned to the Julia garbage collector for the mark phase.

4.3.5 (Mark) Garbage Collector Scans Through Thread Stacks to Find Roots

Julia iterates through the list of Cilk *pgcstacks* objects to discover all possible roots. The *pgcstacks* are added to the garbage collector stacks in the same manner as for regular Julia threads. Julia tasks created the roots on Cilk stacks so Julia knows how to mark and sweep its own *pgcstacks* from Cilk workers. The garbage collector will be able to unpack all of these *pgcstacks* to discover all possible roots.

4.3.6 (Sweep) Reclaim Space Occupied by Unmarked Objects

Now that Julia has all necessary *pgcstacks* from the Cilk runtime, garbage collection can occur as normal. The mark and sweep phases occur on all discoverable roots in the Julia threads and Cilk workers. The discoverable roots are in 3 aspects of the program: Julia *pgcstacks*, Cilk workers *pgcstacks* from the reducer, and the *pgcstacks* inside of suspended Closures.

4.3.7 Threads Resume as Before

Garbage collection ends and all threads are sent a signal to continue execution as before. The runtime is back to regular operation. Cilk workers and Julia threads can continue executing as before, now with less garbage on their heaps.

Chapter 5

Analysis

In this section I will be providing analysis on the Julia and OpenCilk integration. In design of the integration system, a few aspects appeared to be vital. OpenCilk was designed to be a modular system in the compiler and runtime. One major goal was to be able to integrate OpenCilk and Julia with minimal changes to either system. After a completion of the integration, it is worthwhile to go through and evaluate different aspects of the design and determine of the original goals were achieved.

Changes were made in 3 main components: the Julia codebase (runtime and compiler), Opencilk codebase (runtime), and glue code akin to a library but currently exists inside of the Julia runtime. A majority of the changes were in the glue code that currently lives alongside the Julia runtime. In the following sections I will break down each of the parts that were changed and evaluate the changes to assist future runtime combinations. This thesis is meant as an exploration and case study in runtime integrations of OpenCilk, Julia, and future languages.

5.1 Glue Code

The glue code was designed to be similar to a library which Julia users could choose to use or to leave out. The goal is to have a majority of the changes inside of the glue code.

Inside of the glue code are multiple parts:

1. The Cilk reducer to store all stack allocated roots inside the Cilk workers
2. Cilk reducer functions and definition
3. *pgcstack* lookup functions to determine if the work executed is on a Cilk worker or Julia thread for LLVM
4. Cilk initialization code to create the *ptls* variables on each of the Cilk workers correctly
5. Cilk closure iterator
6. Cilk *pgcstack* linked list discovery through a Cilk reducer
7. Cilk stop the world functionality
8. Cilk initialization callback
9. Cilk garbage collection safepoint callbacks

All of these changes are vital to the integration. The proposed changes have a majority of the code inside of the glue code to achieve the goal of minimal changes to either of the existing runtimes. The glue code sits between the two runtimes to describe interactions between them.

5.2 Cilk Minimal Code Change

In this section I will evaluate the changes made to the OpenCilk runtime. These changes were designed to be minimal to allow easier incorporation between the two systems.

The main changes in the Cilk runtime were the initialization of the system, entering a worker stealing section callback, and entering thief section callback. The OpenCilk runtime has updates to allow Julia to register reducers with the Cilk runtime and to add initialization functions during startup. The functionality allows the Cilk runtime to incorporate easier with the Julia runtime. There are 3 callbacks

added to Cilk to complete the integration. The startup callback to initialize the workers correctly, a callback to change the *ptls* of a Cilk worker when becoming a thief, and a callback to change the *ptls* of the Cilk worker when starting to execute work (leaving the work-stealing loop). The callback design previously existed in Cilk to add functions which ran at the startup of the runtime or the termination of the runtime. The integration code in Cilk adds callbacks functionality to add foreign code without adding any Julia specific code itself. The OpenCilk codebase has changed around 50 lines total in the integration work.

In addition to changes meant for the Julia integration, the Cilk runtime now has new methods for:

1. Iterating over all worker reducer views of a system given the workers. Cilk did not have a mechanism for discovering all current worker views of a reducer without the reducer executing the code. For example, if worker 1 wanted to see their local view of a variable inside a reducer it could use the $REDUCER\_VIEW$ function; however, if the Cilk runtime desires the local view of worker 1's variable there was no functionality to do so. Previously, the main way to access a reducer view of a specific worker was to be executing on the worker itself. The Julia garbage collector requires the views of Cilk workers without actaully executing code from this worker. A new function was created which allows the Cilk runtime to iterate over all views of Cilk workers. The feature was necessary to find any live *pgcstacks* in the system but the addition can be used for future work with Cilk. Of note: the new view of every worker is not thread-safe; in the Julia integration every worker is in a stop the world state during garbage collection and was not a consideration for this thesis. One should be careful using the lookup of a reducer view on a worker when the view could be changing if the worker is executing work.

2. Another addition to the Cilk runtime is the ability to scan through the Closure tree starting from the root Closure. The integration design requires a scanning of all Closures in the Closure tree. These contain all relevant *pgcstacks* for the

Julia integration, but are not specific to the integration. The functionality was not built into Cilk but was added with this thesis. The code is useful for future work with Cilk to allow the scanning of internal data structures.

These changes to the Cilk runtime are minimal. The new functionality is also a benefit to the overall Cilk system because they are useful for future applications.

5.3 Julia Code Changes

In this section I will evaluate the changes made to the Julia runtime. The changes involve using the glue code to track the Cilk runtime during execution. The glue code currently exists within the Julia runtime source code, however it was designed to be moved out of Julia and used as a library. The changed sections are listed below:

1. Garbage collection iterates over *pgcstacks* in the Cilk runtime used by Cilk workers
2. Garbage collection calls to the glue code to retrieve Cilk stacks
3. Lookup of Cilk worker *pgcstack* during garbage collection
4. Callbacks into the OpenCilk runtime to discover Cilk state
5. LLVM compilation changes calling a *pgcstack* of Cilk workers or Julia threads
6. Compiler updates to include changes from OpenCilk

These changes are necessary to allow the combination to exist. The change in the system to use an extra runtime with glue code and compiler support requires there to be changes in the central runtime. While there are multiple locations that code was changed, many of these areas was expanding on the currently used functionality within the Julia runtime. For example, after the glue code returns a list of *pgcstacks* the garbage collector must iterate over, the existing Julia functionality adds these *pgcstack* pointers to the garbage collector. While these changes are Julia specific, the methodology for combining OpenCilk with a precise garbage collector is not.

5.4 Speed Evaluation

In this section I will evaluate the changes as it pertains to compilation and execution speed. The proposed changes to the Julia runtime should not effect the performance of the overall system as a design goal.

5.4.1 Compilation Time

The proposed changes currently have an impact on the compilation of the Julia system. The compilation times are slower likely because of the LLVM lowering step that add extra time to the compilation of Julia. The lowering step is forking to decide if the *pgcstack* should be evaluated from the Cilk worker or the Julia thread which is currently believed to be the unoptimized slowdown. This is an area of continued research to detect the actual increase in compilation time and discover a fix to the issues.

5.4.2 Execution Time

The integration requires more development to be able to test execution time. When completed, the benchmarks of the integration should resemble the speed of their respective languages. The OpenCilk workers, excluding garbage collection time, should see minimal performance degradation to using OpenCilk workers outside of Julia. Julia's garbage collection execution time should be similar to Julia without the integration but with the added time of scanning extra stacks on the Cilk workers.

Chapter 6

Related Work

In this chapter I will discuss how other languages handle parallelism and previous work in the combination of runtimes.

6.1 External Objects in Julia

Julia contains functionality to add external objects to the runtime. These can be used as roots to objects the a program and garbage collected within Julia's garbage collector. The method is used for some libraries to enable programmers to extend the Julia language with new objects without writing a new garbage collector. GAP is a Julia extension which utilizes the functionality. GAP [13] uses external objects to Julia, creates the objects in the Julia runtime and provides callback functions to the garbage collector for mark and sweep. The mark phase is required, by Julia, to mark all objects still in use at the program and sweep defines how the objects can be swept away. The functionality allows GAP to act as an extension to the runtime/garbage collector. The approach is useful because GAP can create a library for Julia with external types that are managed by Julia's built in systems. While executing code for the GAP extension, Julia does not understand what the new foreign types are but is supplied with enough detail to garbage collection them. The initial approach to this thesis was to use Julia's external garbage collector module similar to GAP. One downside of this approach was having to re-write systems already in use by

Julia on Cilk workers. The Julia runtime is sending work with predefined objects to the Cilk runtime so while the objects may not be allocated on Julia threads, the runtime has inclinations about the types of objects and their state data in *pgcstacks*. In a situation where external objects were necessary to the runtime combination, the external object allocator Julia features come in useful. The minimal extension approach proposed in this thesis is more integrated with Julia: the runtime creates roots on Cilk's stacks and already knows how to treat it (there are no external types). The system proposed is a closer integration to allow Cilk to be shipped with Julia.

6.2 MPI OpenMP

One attempt at combining two runtimes for parallelization is MPI [7] and OpenMP [3]. MPI is a message passing interface which can work on shared distributed memory. OpenMP is an implementation of a multithreading interface used commonly for parallel computing. Inside of OpenMP, there exists a primary thread which forks a specific number of sub-threads. These receive tasks to be executed on top of them. Inside of the combination of MPI and OpenMP [26], there is a way where the two are able to be used together as a hybrid system. The combined system often suffers from poor scaling with load imbalance, too fine a grain problem size, memory limitations, and poor optimization [18]. The over-subscription of processor cores causes the overall program to be sometimes slower than doing just OpenMP or MPI separately. In the work done by Capello et al., the authors discovered the parallel processing benefits depended largely on the hardware system and the level of shared memory in the system. In the combination of MPI and OpenMP, the user must be knowledgeable about the type of system and have a large amount of knowledge in how to write efficient parallel code. Over-subscription can lead to worse performance in a combined parallel system than using MPI alone.

Another attempt [22] at creating a parallel version of MPI uses pthreads as the parallelism framework. The authors created an algorithm for efficiently spawning threads to enable intra-node and inter-node balancing of work. The efficiency is

again architecture dependent requiring the programmer to have knowledge about the system reducing the portability of the system. The Cilk-Julia integration tries to avoid the dependency of a programmer to have a vast knowledge of writing efficient parallel programs. The fine-grain nature of work stealing in OpenCilk achieves this goal.

6.3 Parallel Machine Learning Frameworks

Another set of systems combinations with parallel code are machine learning frameworks called from user code. Some machine learning frameworks assume they have full control over the system when they receive work. This works great for performance if the framework is exclusive; however, if the program continues executing work alongside the framework execution then the system fights for computation power. In addition, a system may have work from other programs executing work at the same time on a machine. When systems compete, parallel partitions of work will have to wait while others execute. A lack of dynamic scheduling or balancing causes the different programs to fight over the resources. TensorFlow [14] is an example of a machine learning framework utilizing pthreads. When training is occurring, oversubscription of threads can occur leading to performance degradation [28]. This is a situation where a fine-grain parallelism framework such as Cilk would be beneficial: threads are not explicitly tied to work and users create areas where threads can take advantage of parallel execution to offer performance in practice. The Cilk-Julia combination presented in this thesis strives to achieve an efficient threading model which scales with amount of work. If all work is done inside of Cilk-Julia in parallel then the system will not get oversubscribed because OpenCilk's scheduler will be aware of other processes occurring.

Chapter 7

Conclusion

This thesis presents a case study on combining two runtimes. The OpenCilk and Julia integration is elegantly created to offer users a simple abstraction to show the Julia compiler where parallelism exists within a program. The integration of the runtimes is minimal and utilizes many of the existing Julia and OpenCilk interfaces. This chapter includes contributions and future avenues of work for the integration.

This work makes several contributions to the runtime combination of OpenCilk and Julia. I demonstrated an approach to combining two different runtimes and allow them to work together. A novel algorithm was developed for managing Julia roots on OpenCilk's worker cactus-stack. The algorithm depends on changes in the Julia runtime, OpenCilk runtime, and combination glue code. Specifically, a reducer hyperobject is used to manage the stacks on Cilk workers at all times. This hyperobject allows the Julia garbage collector to discover roots of a program on Julia threads and Cilk workers. I created a new method for managing objects on a Cilk stack by utilizing the Julia methods that already exist within the runtime. While aspects of the Julia and OpenCilk combination are solved, there are more avenues that need to be explored. My work leaves several continuation points for future work.

7.1 Future Work

Some aspects of the the runtime integration need to be completed. Parts of the combination are not working correctly such as finalizers inside of Julia. The Julia runtime creates the ability to run code on top of the garbage after each garbage collection run, known as a finalizer. In the current integration finalizers have been removed and need to be worked on more. There are extra steps that need to be accounted for when running a finalizer on a Cilk stack to coordinate.

Another aspect requiring more work to complete the integration is to have $world-age$ inside the $ptls$ work with Cilk workers. The age is required for aspects of the language and is currently not tracked correctly when executing on Cilk workers. This is a state of the thread-local storage which requires more exploration. It may need to exist in a separate reducer or more logic could be necessary in the Cilk runtime. An update to Julia is moving the $world - age$ out of the $ptls$ in the future. When the update occurs, the integration will be required to update also.

Bibliography

[1] C library function - longjmp() - Tutorialspoint. https://www.tutorialspoint.com/c_standard_library/c_function_longjmp.htm, (accessed 2021-05-19).

[2] The Go Programming Language. https://golang.org/, (accessed 2021-05-19).

[3] Home. https://www.openmp.org/, (accessed 2021-05-19).

[4] Java | Oracle. https://www.java.com/en/, (accessed 2021-05-19).

[5] The Julia Programming Language. https://julialang.org/, (accessed 2021-05-19).

[6] The LLVM Compiler Infrastructure Project. https://llvm.org/, (accessed 2021-05-19).

[7] MPI Forum. https://www.mpi-forum.org/, (accessed 2021-05-19).

[8] Multi-Threading · The Julia Language. https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.@spawn, (accessed 2021-05-19).

[9] OpenCilk. https://cilk.mit.edu/, (accessed 2021-05-19).

[10] pthreads(7) - Linux manual page. https://man7.org/linux/man-pages/man7/pthreads.7.html, (accessed 2021-05-19).

[11] Welcome to the libuv documentation — libuv documentation. http://docs.libuv.org/en/v1.x/, (accessed 2021-05-19).

[12] OpenCilk/cheetah. May 2021. https://github.com/OpenCilk/cheetah, (accessed 2021-05-19).

[13] oscar-system/GAP.jl, April 2021. https://github.com/oscar-system/GAP.jl, (accessed 2021-05-19).

[14] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. *arXiv:1605.08695 [cs]*, May 2016. arXiv: 1605.08695.

[15] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017.

[16] Robert D Blumofe. Scheduling Multithreaded Computations by Work Stealing. page 29.

[17] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. page 11.

[18] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *ACM/IEEE SC 2000 Conference (SC'00)*, pages 12–12, Dallas, TX, USA, 2000. IEEE.

[19] Ulrich Drepper. ELF Handling For Thread-Local Storage. page 81, 2013.

[20] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 79–90, New York, NY, USA, August 2009. Association for Computing Machinery.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):212–223, May 1998.

[22] Juan F.R. Herrera, Leocadio G. Casado, Remigijus Paulavicius, Julius ilinskas, and Eligius M.T. Hendrix. On a Hybrid MPI-Pthread Approach for Simplicial Branch-and-Bound. In *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, pages 1764–1770, May 2013.

[23] I-Ting Angelina Lee, Silas Boyd-Wickizer, Zhiyi Huang, and Charles E. Leiserson. Using memory mapping to support cactus stacks in work-stealing runtime systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, page 411, Vienna, Austria, 2010. ACM Press.

[24] I-Ting Angelina Lee and Tao B. Schardl. Efficiently Detecting Races in Cilk Programs That Use Reducer Hyperobjects. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures*, SPAA '15, pages 111–122, New York, NY, USA, June 2015. Association for Computing Machinery.

[25] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–265, Austin Texas USA, January 2017. ACM.

[26] Lorna Smith and Mark Bull. Development of mixed mode MPI / OpenMP applications. *Scientific Programming*, 9(2,3):83–98, 2001. Publisher: IOS Press.

[27] Robert Utterback, Kunal Agrawal, Jeremy T. Fineman, and I-Ting Angelina Lee. Provably Good and Practically Efficient Parallel Race Detection for Fork-Join Programs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 83–94, New York, NY, USA, July 2016. Association for Computing Machinery.

[28] Xinchen Wan, Hong Zhang, Hao Wang, Shuihai Hu, Junxue Zhang, and Kai Chen. RAT - Resilient Allreduce Tree for Distributed Machine Learning. In *4th Asia-Pacific Workshop on Networking*, APNet '20, pages 52–57, New York, NY, USA, August 2020. Association for Computing Machinery.