Function-level Parallelism for Multi-core Processors
Automated Parallelization of Graphical Dataflow Programs

by

John Vilk

A Major Qualifying Project report submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science
in
Computer Science
at Worcester Polytechnic Institute
2010

Advisors:

Professor Craig Wills, Advisor
Professor Hugh Lauer, Co-advisor

TABLE OF CONTENTS

ABSTRACT 5

CHAPTER

I. Introduction 1

1.1 Background 2

II. Graphical Dataflow Language & Translating to C 5

2.1 Introduction to Graphical Dataflow Languages 5
2.1.1 Datatypes 6
2.1.2 Operations 8
2.1.3 Loops 12
2.1.4 Scopes 16
2.1.5 Conclusion 16
2.2 Translating Graphical Dataflow Programs to C 17
2.2.1 Datatypes 18
2.2.2 Operations 19
2.2.3 Scopes 21
2.2.4 Loops 22
2.2.5 `main` Method 24

III. Methodology 25

3.1 Activation Records 27
3.1.1 Traditional Activation Records 27
3.1.2 Our Activation Records 28
3.1.3 Allocating Activation Records 29
3.1.4 Instantiating Activation Records 31
3.1.5 Suspending and Resuming Activation Records 34
3.2 Program Transformations 40
3.2.1 Function Entry Point 40
3.2.2 Establish Function Argument Order 41

3.2.3 Entering a Scope 42
3.2.4 Writing Arguments into Activation Records 47
3.2.5 Waiting For Return Values 50
3.2.6 `for` Loops 51
3.2.7 `foreach` Loops 52
3.2.8 Function Termination 52
3.2.9 Late Compilation Changes 52
3.3 Runtime Library 55
3.3.1 Allocating Activation Records 55
3.3.2 Ready Queue 55
3.3.3 Resuming Activation Records 55
3.3.4 Thread Pool 56

IV. GCC Background 57

4.1 Language Front End 57
4.2 GENERIC Trees 58
4.3 GIMPLE 58
4.3.1 High GIMPLE 59
4.3.2 Low GIMPLE 59
4.3.3 SSA GIMPLE 60
4.4 RTL 61
4.4.1 Nonstrict RTL 61
4.4.2 Strict RTL 62
4.5 Assembly 62

V. GCC Plugin Implementation 63

5.1 GCC Plugin Architecture 63
5.2 GENERIC Tree Hook 64
5.3 High GIMPLE Pass 65
5.3.1 Start of Function Label 65
5.3.2 Local Variable Representation of Activation Record Elements 65
5.3.3 Allocating and Initializing Activation Records . . . 66
5.3.4 Adding Constants to Counter 66
5.3.5 Writing Arguments into Activation Records 67
5.3.6 Waiting for Return Values 67
5.3.7 Function Termination 67
5.4 Strict RTL Pass 68
5.4.1 Suspend/Resume Code 68
5.4.2 Aligning Activation Records 69

VI. Conclusion 70

6.1 Conclusion 70
6.2 Future Work 71
6.2.1 Shifting from GCC 71
6.2.2 Loops 72
6.2.3 Runtime Library 72
6.2.4 Graphical Dataflow Program Development Environment 72

BIBLIOGRAPHY 73

ABSTRACT

Function-level Parallelism for Multi-core Processors
Automated Parallelization of Graphical Dataflow Programs

by

John Vilk

Advisor: Professor Craig Wills

Co-advisor: Professor Hugh Lauer

This project investigated how to implement implicit fine-grain parallelism in graphical dataflow programs at the function-level. We succeeded in developing a runtime model that accomplishes this by changing the content of each function's activation record, and changing its location from the stack to the heap. Simple atomic instructions are used for synchronization purposes, and a runtime library manages a thread pool that it uses to execute functions to minimize operating system calls. We call our modified functions *concurrent function invocations*. In addition, we describe how to implement this runtime model as an optimization pass within a compiler that would work on a well-defined C representation of graphical dataflow programs. Finally, we describe how this optimization pass might be implemented in GCC, which opens the door to future work that could complete an implementation of our runtime model.

CHAPTER I

Introduction

For over 40 years, Moore's Law has greatly influenced processor manufacturers to rapidly increase the performance of their products. In accordance with this, the clock speed of retail processors has exponentially increased during much of this time. However, in recent years, power dissipation issues have prevented the companies from making further large gains in clock speed. To keep up with Moore's Law, these companies made a dramatic shift to multi-core architectures. Today, multi-core processors are present in most retail computers, and the number of cores in each are increasing rapidly.

However, many programs are still written as single-threaded applications, and do not take advantage of the multiple cores available to them. Those that *are* written to take advantage of multiple cores generally take a coarse-grain approach to parallelism, and assign significant amounts of work to each parallel section. Typically, the result is a few important threads running at once, when modern processors are capable of executing many more simultaneously.

Fine-grain approaches to parallelism seek to maximize the use of parallel processing by executing many small tasks in parallel in order to use as many cores as possible. A program written in this manner would, theoretically, see improved performance as the number of cores on processors increases. Unfortunately, it is very difficult to break

up a program into a large number of tasks that can execute in parallel. In addition, this approach leads to a lot of operating system overhead through thread creation and scheduling these parallel tasks for execution.

Graphical dataflow programs have specific properties that lend themselves to fine-grain parallelism. Specifically, dependencies among functions in graphical dataflow languages are very explicit because they are expressed in the form of arguments and return values. These dependencies could be used to automatically determine when and where to execute functions in parallel, thus relieving the burden of breaking up a program into parallel tasks from the programmer. The overhead of scheduling parallel tasks and thread creation can be mitigated using runtime support that manages a thread pool. This project aims to explore how to implement implicit fine-grain parallelism in graphical dataflow programs at the function level.

1.1 Background

There are a number of shortcomings to traditional tread-level parallelism that prevent it from being feasible at the fine-grain level. First, there is overhead involved when creating a thread or process, since it requires a call into the operating system. Since fine-grain parallelism parallelizes small tasks, the time to create the thread or process may be significant when compared to the time to execute the task. Second, all threads or processes must be synchronized; as the number of threads increases, so does the amount of synchronization required. Third, traditional programming languages (e.g. C, C++, Java) require the programmer to explicitly handle synchronization, communication, and race conditions, so implementing a program with a large number of parallel tasks would require skillful programming to avoid common pitfalls. Finally, scheduling becomes an issue, since the program will want to run enough fine-grain tasks to take full advantage of each core on the processor, but not too many such that the cores are swamped with parallel tasks. There have been a number of attempts to

mitigate some of these shortcomings, which we will describe below.

In 2007, Intel published a paper titled "Architectural Support for Fine-Grained Parallelism on Multi-core Architectures" that described how to overcome scheduling and thread creation overhead by moving those responsibilities into the processor (1). They implemented hardware queues to cache tasks and handle task scheduling policies, and task prefetchers on each core to decrease the latency of accessing the hardware queue. The result is a system where the overhead of task scheduling is minimal, and the execution speed is nearly ideal. On a 64-core machine, their task benchmark ran 98% faster than a pure software solution. However, this solution requires specialized hardware that is not yet readily available. It also requires the programmer to explicitly implement fine-grain parallelism using a task queue API.

The next version of Java, Java 7, introduces a package called `forkjoin` that allows fine-grain parallelism in the form of tasks (2). Tasks are much more lightweight than traditional threads, since they do not need to perform calls into the operating system during creation. Rather, `forkjoin` maintains a thread pool of "worker" threads that only need to be created once, and can be reused for as many tasks as needed. The `forkjoin` package also takes care of thread scheduling, as its thread pool has as many threads as cores on the CPU. However, like in the Intel solution, the programmer must explicitly create these tasks.

A previous MQP successfully implemented implicit fine-grain parallelism for a graphical dataflow language. The project developed a system that enabled programmers to construct graphical dataflow programs and translate them into a parallelized C++ form that could be compiled with a C++ compiler. Like with Java 7's `forkjoin` package, this implementation used a thread pool to minimize thread creation events. Unfortunately, it used an antiquated programming language called SISAL to enable parallelism, and the system they developed had stringent limitations on the sorts of programs that could be created (3).

Using the infomation gathered from these sources, we made a few key decisions for our project. First, function scheduling and thread pool management would be delegated to a *runtime library* that runs in user space. In the future, processors may have direct support for task queues, like in the previously mentioned Intel paper (1). Separating the runtime library from the rest of the parallelization process allows these platforms to have a separate implementation that takes advantage of these hardware features. For other architectures, a purely software implementation of the runtime library can manage a thread pool, much like Java 7's `forkjoin` package. The implementation of this runtime library is outside the scope of this project, but we will describe its responsibilities and features. Second, to avoid the trap that the previous MQP fell into with SISAL, we decided to look into how to implement implicit fine-grain parallelism within GCC, an open source compiler framework that is widely used and is under active development (4).

To achieve these goals, we have to convert graphical dataflow programs into a representation that can be understood by GCC and perform program transformations on this representation within GCC to introduce parallelism at the function level. Chapter II describes how we solved the first problem with a process to translate graphical dataflow programs into a stylized subset of C that can be run through the GCC compiler and executes serially. Chapter III explains the runtime model we developed to support fine-grain parallelism at the function level. Chapter IV describes GCC's compilation process for important background information. Finally, Chapter V provides an overview of how to implement the runtime model from Chapter III as a GCC plugin.

CHAPTER II

Graphical Dataflow Language & Translating to C

In this chapter, we outline a generic graphical dataflow language inspired by LabVIEW. Then, we describe how to convert programs written in this graphical dataflow language into C for the parallelization process described in Chapter III.

2.1 Introduction to Graphical Dataflow Languages

Graphical dataflow languages are unlike most traditional programming languages. While traditional programming languages are written entirely in a textual form, graphical dataflow programs take the form of a directed graph. Nodes in the graph represent functions, conditionals, variables, and constants. Edges represent data connections. In this chapter, we will introduce a generic graphical dataflow language inspired by LabVIEW (5). Programs written in this graphical dataflow language can be automatically parallelized using the parallelization process described in Chapter III.

To get our feet wet, let's look at a simple function written in this language. An example can be seen in Figure 2.1. In this example, the name of the function is `f`, as indicated by the text above the rectangle that encloses the nodes of the graph. This rectangle represents the **scope** of `f`, just like curly braces represent scopes in C. The two boxes labeled *int* are called *terminals*, and are *outside* the scope. Terminals

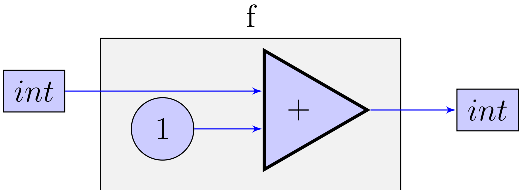


Figure 2.1: A very simple graphical dataflow function, `f`, that adds 1 to the input number.

represent input and output in a component, and serve the role of parameters and results of textual languages. The input is supplied by the user if the function is run directly, or by another function, and serves the role of an argument. The output is sent to the user if the function is run directly, or it is sent to another node in a graph that contains a call to the function. The triangle with the `+` symbol represents the addition operation. The circle with `1` represents the constant 1. According to the edges, the function's input and the constant 1 are the operands to the `+` operation, and the output of the operation is also the output of the function `f`. From this, we can deduce that this function's purpose is to add 1 to the input number. In this language, data flows along the links, and operations at nodes execute when the data values have arrived.

In the sections below, we briefly document the features of this graphical dataflow language to provide important background information needed to fully understand the rest of this report.

2.1.1 Datatypes

Our graphical dataflow language has no concept of variables; instead, data travels around in colored edges, or *wires*. The color and style of the wire represents its datatype. There are no pointers or references in this language; all data are passed by value. Below, we outline the data types available in our graphical dataflow language.

2.1.1.1 Boolean

Boolean

A *Boolean* is represented by a solid green line. Booleans are either `true`, or `false`.

2.1.1.2 Integer

Integer

An *integer* is represented by a solid blue line.

2.1.1.3 Floating Point

Floating Point

A floating point number, or *float*, is represented by a solid orange line.

2.1.1.4 Arrays

1	Boolean	Integer	Floating Point
Base Type			
1D Array			
2D Array			
3D Array			

Figure 2.2: The above table displays a variety of array types along with their base types.

The graphical dataflow language supports arrays of any base type. The base type determines the color of its representative wire, while the dimension of the array determines the multiplicity of the wire. These types can be seen in the table in Figure 2.2. A one-dimensional array is a thick line, a two-dimensional array is a double line, a three-dimensional arrays is a triple line, etc. Each array is also aware of its size, which may change dynamically.

Multi-dimensional arrays with dimension d contain elements that are arrays with dimension $d - 1$. For example, the elements of a two-dimensional array are one-dimensional arrays.

2.1.2 Operations

2.1.2.1 Basic Arithmetic

Operation	**Representation**	**Result**
Addition	a b + c	$c = a + b$
Subtraction	a b - c	$c = a - b$
Multiplication	a b × c	$c = a * b$
Division	a b ÷ c	$c = \frac{a}{b}$
Modulus	a b % c	$c = a\%b$

Figure 2.3: Arithmetic operations in the graphical dataflow language.

The graphical dataflow language supports basic arithmetic operations, such as addition, subtraction, multiplication, division, and modulus. They operate on `int` and `float` datatypes. These operations are represented as triangles with the operator printed inside. A table of operations and their representations in the language can be seen in Figure 2.4.

Operation	Representation	Valid Input Types	Result
Less Than	a, b → $<$ → c	`float`, `int`	$c = a < b$
Less Than Or Equal To	a, b → $\leq$ → c	`float`, `int`	$c = a \leq b$
Greater Than	a, b → $>$ → c	`float`, `int`	$c = a > b$
Greater Than Or Equal To	a, b → $\geq$ → c	`float`, `int`	$c = a \geq b$
Equality	a, b → $=$ → c	`float`, `int`, `bool`	$c = a == b$
Logical And	a, b → $\&$ → c	`bool`	$c = a\&b$
Logical Or	a, b → $\vert$ → c	`bool`	$c = a\vert b$

Figure 2.4: Comparison operations in the graphical dataflow language.

2.1.2.2 Comparison Operations

Comparison operators compare two input values, and output a Boolean with the result of the comparison. A table of the comparison operators in our graphical dataflow language can be seen in Figure 2.4.

2.1.2.3 Conditional Switch

The graphical dataflow language has a conditional operation called a *conditional switch*. Much like an `if-else` or `switch` statement in other languages, it presents a branching path in the program for different values of a variable, and it defines a

new scope. Unlike traditional `switch` statements, conditional switches must define outputs, and each case *must* supply a value for each output. Conditional switches can also have multiple inputs beyond the input that it is switching on.

A conditional switch is represented graphically as a multi-layered scope with terminals for input and output. Each layer is another case in the switch. All conditional switches must have a default case.

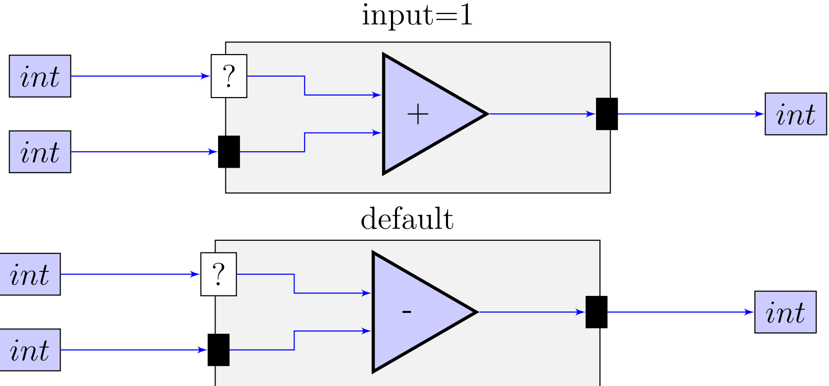


Figure 2.5: A simple conditional switch on an integer. The top scope executes when the input integer is 1, and the bottom scope executes for any other input integer.

Figure 2.5 displays a simple conditional switch on an integer. The switch checks the integer that is connected to the ? box. If the integer is 1, then the switch adds that integer to the other integer input, and outputs the result. If the integer is anything else, then the default case executes, which subtracts the second input integer from the switched on integer, and outputs the result.

2.1.2.4 Function Call

One of the key differences between a graphical dataflow language and languages like Java and C is that a function's output can be "wired" to the input of another. In fact, each argument to a function can come from the output of a different function! By comparison, in C or Java, all arguments to a function must be supplied directly by the calling function. In a graphical dataflow program, there is no concept of a 'calling'

function; a function executes, or *fires*, once it receives all of its inputs. Graphical dataflow functions can also have multiple outputs, and each separate output can be delivered to multiple destinations.

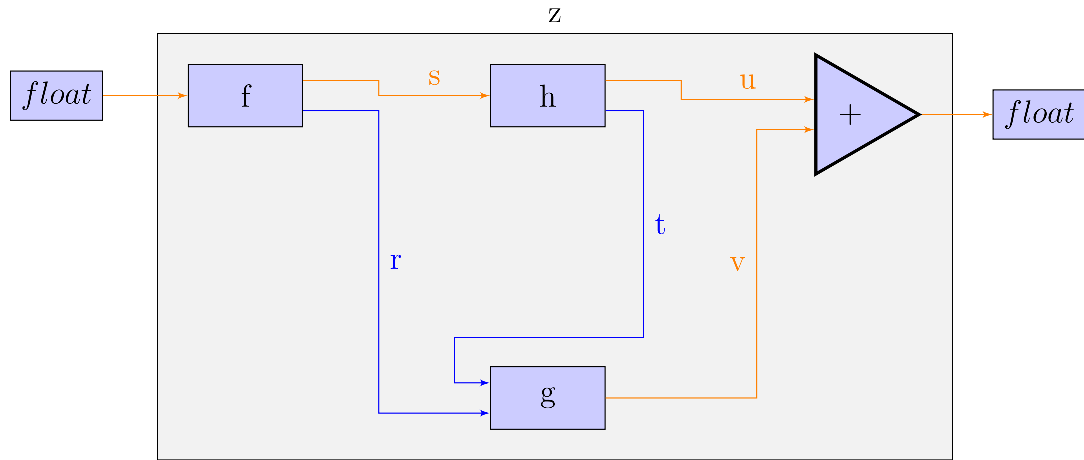


Figure 2.6: A graphical dataflow function scope with multiple function calls.

Figure 2.6 displays a function `z` that illustrates these properties. After `z` receives its input, `f` begins executing because `z`'s input is wired directly into `f`. Function `g` cannot execute until it receives input from both `f` and `h`. Meanwhile, `h` will start executing as soon as `f` outputs `s`. There may be no circular dependencies (like in the invalid function shown in Figure 2.7), so the function `z` can complete successfully in the following order: `f`, `h`, `g`, and, finally, the `+` operation.

Graphical dataflow programs are not forced to execute serially, however. In the previous example, dependencies forced serial execution. In Figure 2.8, the function `z` can execute in multiple valid sequences. Once `f` finishes executing, both `h` and `g` are ready to run. It does not matter which executes first. In fact, they could execute in parallel! It is due to this that we believe that graphical dataflow programs exhibit **inherent parallelism**. Introducing parallel execution at the granularity of a function call is the main idea behind our parallelization technique.

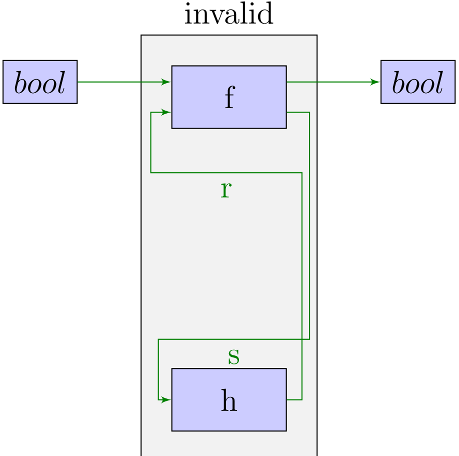


Figure 2.7: An invalid graphical dataflow function with circular dependencies.

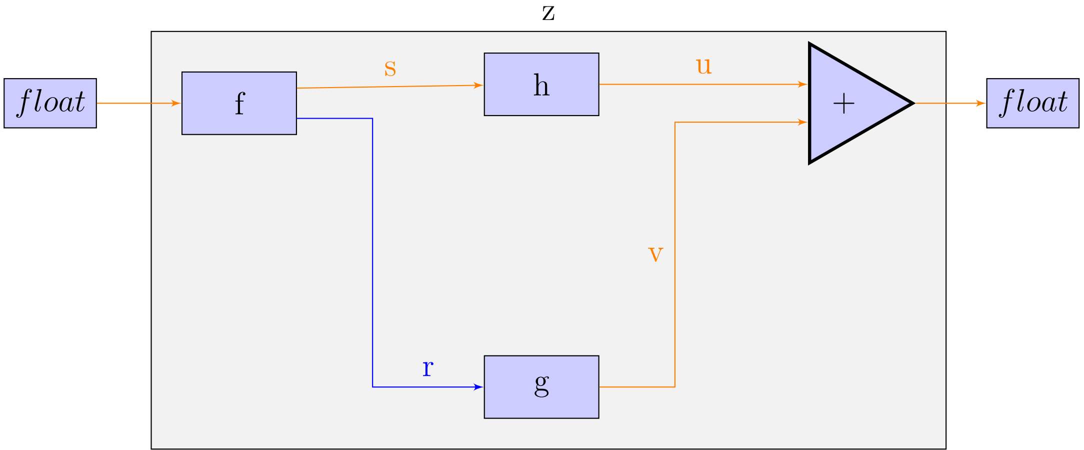


Figure 2.8: A graphical dataflow function scope that does not force a serial execution.

2.1.3 Loops

There are three types of loops in this graphical dataflow language: `for` loops, `foreach` loops, and `while` loops.

2.1.3.1 `for` Loops

`for` loops are loops that iterate a specified number of times. They may include the following items:

- **Input Values and Arrays**: The loop can use these input items in its operations.

- **Output Values or Arrays** : Output values and arrays are produced by the last iteration of the loop.

- **N** (required): N is the upper bound on the number of iterations the loop will perform. N can be set to the size of the smallest input array.

- **i** (required): i is the counter that represents the current iteration of the loop.

- **Disaggregator**: The disaggregator takes in an input array, and, for each iteration of the loop, provides the corresponding element of the array as input to that iteration.

- **Aggregators**: An aggregator creates a new array from the output values of each iteration.

- **Accumulators**: An accumulator performs a *commutative* operation on the output it receives from each loop iteration (e.g. sum all loop outputs). There may be one or more accumulators on a `for` loop. Accumulators are inspired by the programming language Fortress, which has `for` loops that are parallel by default (6).

- **Recirculating Values**: These output values are calculated each iteration and supplied as inputs to the next iteration. There may be one or more recirculating values on a `for` loop.

- **Stop Conditions**: A condition that, if met, halts the loop prematurely. There may be more than one stop condition.

The number of iterations that a `for` loop performs is the minimum of the size of the smallest input array to a disaggregator, and the value of N. N is either explicitly set to a constant, or is set to the size of the input array. If there is no input array, N *must* be explicitly set. If a stop condition is present, a Boolean must be wired to it.

If the Boolean has the value `true` during an iteration, then the loop stops executing after that iteration completes.

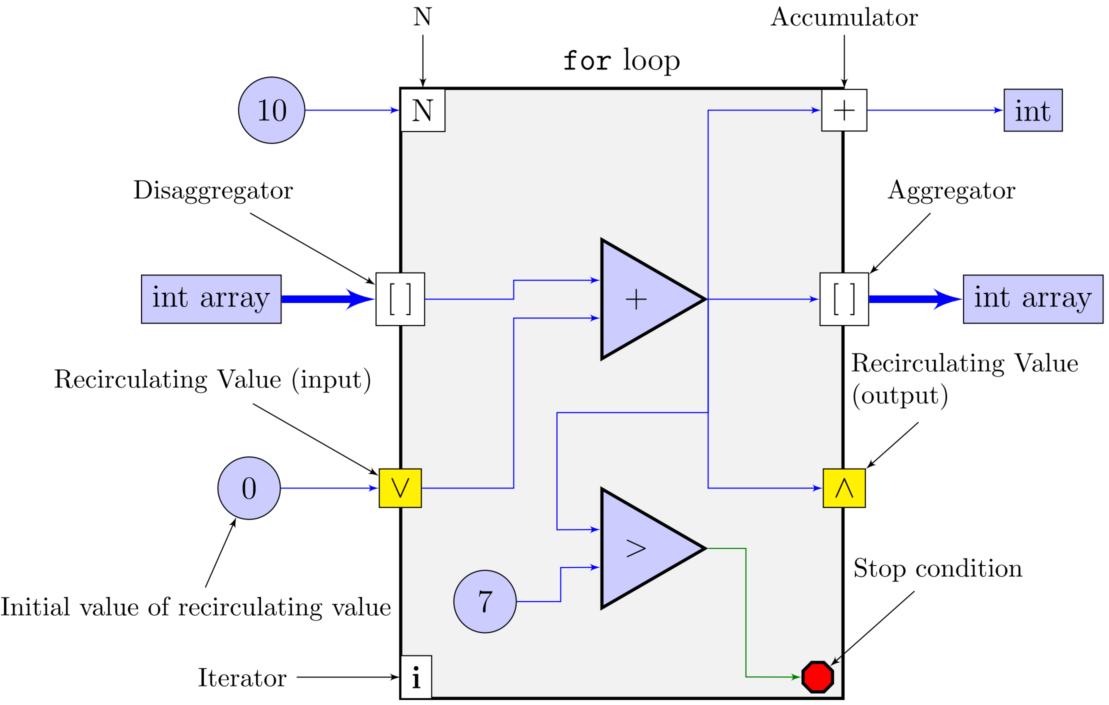


Figure 2.9: This `for` loop uses most of the loop constructs.

A labeled `for` loop that uses most of these loop constructs can be seen in Figure 2.9. **N** is set to 10, so the loop will execute 10 times *unless* the input array has a size that is smaller than 10. During every iteration, the ith element is added to the previous element. The result of this operation is wired to four locations:

- The result is wired into an aggregator that places it into the ith position of the output array.

- The result is wired into an addition accumulator that adds it to its current value. When the loop stops executing, it will output the sum of all of the elements in the output array.

- The result is wired into a recirculating value, so that it can be used in the next iteration's calculation.

- The result is compared to 7. If it is greater than 7, then the stop condition is met, and the loop stops executing once the current iteration completes.

Thus, for a 6-element input array with values $\{1, 2, 3, 4, 8, 5\}$, this loop would output a 4-element array with values $\{1, 3, 6, 10\}$ and 20 for the addition accumulator. The stop condition is triggered during the iteration that produced 10, so the final two elements in the input array are not processed.

2.1.3.2 `foreach` Loops

`foreach` loops bodies execute a specified number of times. Each execution of the loop body processes one element of the disaggregated input array. `foreach` loops may include any of the items defined for `for` loops, except for recirculating values, stop conditions, output values, and output arrays that are not produced by aggregators. Essentially, `foreach` loops are `for` loops where the executions of the loop body are independent of one another. Due to this, `foreach` loops are inherently parallelizable.

The number of iterations that a `foreach` loop performs is the minimum of the size of the smallest input array, and the value of N. If the value of N is not explicitly declared, then it is automatically set to the size of the smallest input array.

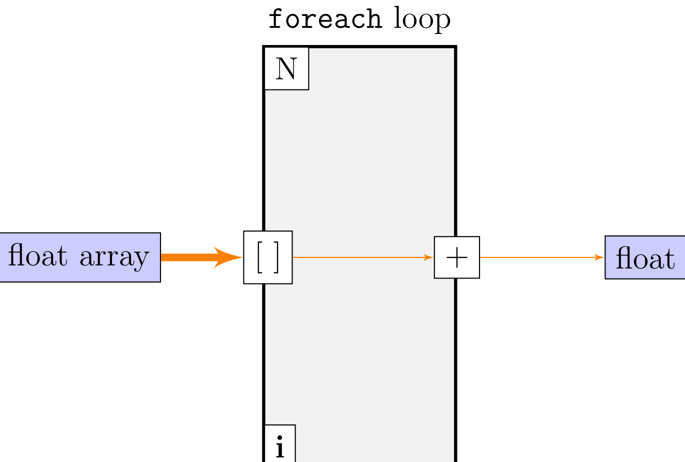


Figure 2.10: A simple `foreach` loop that adds up all of the numbers in the input array.

The `foreach` loop in Figure 2.10 is rather simple: it finds the sum of all of the floating point numbers in the input array using the addition accumulator. N is not explicitly set, so this loop iterates once for each element in the input array.

2.1.3.3 `while` Loops

A `while` loop iterates until a stop condition is triggered. Since they are generally not parallelizable, and are typically used much less often than `for` loops in LabVIEW programs, we will not mention them further in this report.

2.1.4 Scopes

Throughout this chapter, we have mentioned various graphical dataflow language constructs that define "scopes", but we have yet to explain what scopes are in context of our graphical dataflow language.

In a traditional programming language, a scope typically creates a new stack frame to store information local to that scope. However, our graphical dataflow language has no local variables, so this detail is irrelevant to its scopes. Rather, the scopes in our graphical dataflow language are purely lexical scopes, and are mainly important in the under-the-hood operations described in Chapter 3.

In our graphical dataflow language, scopes are defined for:

- Function bodies.
- Loop bodies.
- Branches of conditional switches.

2.1.5 Conclusion

We have described all of the major elements of our generic graphical dataflow language. Now, we will describe how to represent graphical dataflow programs in the

correct format for our parallelization process outlined in Chapter III.

2.2 Translating Graphical Dataflow Programs to C

Unfortunately, there are no compilers for our hypothetical graphical dataflow language, so we need to use an intermediate form that we can parallelize from within an existing compiler. We chose C as an intermediate language for our graphical dataflow language, since it was the simplest vehicle for getting our graphical dataflow programs into the GCC compiler. The translation process outlined in this chapter produces a C program from a graphical dataflow program that can be compiled and run with no modifications. However, the C program will execute in a completely serial manner, with no parallelization at all. The parallelization process described in Chapter III describes how a compiler plugin can modify these C programs from within a compiler to introduce parallelism. Because this parallelization process makes assumptions about the program it is processing, it is important to conform to the conversions explained in this chapter.

To get our feet wet, let's look at the simple graphical dataflow function in Figure 2.1. This function adds 1 to the input number. It would get translated into the following C code:

```
struct f_output
{
  int output;
}

struct f_output f (int input)
{
  //We could also use input++. The compiler looks at both in exactly
  //the same way.
  int output = input + 1;

  struct f_output f_out = { output };

  return f_out;
}
```

In this example, the conversion process is very straightforward, and predictable. We require all functions to return structs. This requirement is explained in Section 2.2.2.3.

2.2.1 Datatypes

The basic datatypes of our graphical dataflow language translate nicely into C. An integer is an `int`, a floating point number is a `float`, and, with the help of `stdbool.h`, a Boolean is a `bool`.

Arrays are a bit more difficult. Unlike C arrays, our arrays must know their size. We can get around this restriction by representing arrays using the following struct, where `T` is the type of the elements in the array:

```
struct T_array
{
  T *array;
  unsigned int size;  //Number of elements in the array.
};
```

Note that these structs **must** follow the above naming convention for the parallelization process to work. Thankfully, the C preprocessor makes it easy to generate code like this; with the following code in a header file, new array types c an be declared with `ARRAY(typename)`:

```
#define ARRAY(T) struct (T)_array \
{ \
  (T) *array; \
  unsigned int size; \
};
```

Notice that the C array in the struct above is referenced by a pointer. Because structs in C must be a constant size, the array information needs to be dynamically allocated and referenced in the struct as a pointer. This poses a problem: due to the semantics of our graphical dataflow language, arrays are passed by value. We have decided that, for the sake of optimization, our implementation will use the C

convention and pass arrays around using a pointer whenever possible. Meaning, if an array is output to only one location, it will be passed as a pointer. If it is output to multiple locations, then it may have to be copied once for each output location.

2.2.2 Operations

2.2.2.1 Basic Arithmetic & Comparison Operations

All of the basic arithmetic and comparison operations in our graphical dataflow language have direct C equivalents. In all cases, we use these equivalent operations.

2.2.2.2 Conditional Switch

Conditional switches are implemented using the `switch` statement. As mentioned before, a conditional switch requires each case to have the exact same outputs, while a `switch` in C does not. This restriction needs to be enforced in the frontend that constructs the graphical dataflow program, because the compiler plugin, in its proposed form in Chapter 3, does not check it.

In addition, each branch of a Conditional Switch defines a new scope in our graphical dataflow language, so the frontend must apply the scope conventions that are described in Section 2.2.3.

As an example, we will convert the conditional switch from Figure 2.5 into a `switch` statement. Because our graphical dataflow language does not have named outputs or inputs, we use arbitrary variable names to represent their values in C:

```
int a, b, output;
//Scope requirement: Assign variables that are inputs to
//the scope into new local variables.
a = var_to_switch_on;
b = other_input_from_scope;

switch(a)
{
case 1:
  output = a + b;
```

```
  break;

default:
  output = a - b;
  break;
}
```

2.2.2.3 Function Calls

Function call translations are best explained using an example. Figure 2.6 shows a graphical dataflow function with multiple function calls. Some of these function calls have multiple outputs that are connected to the input of different functions!

In the C representation, we cannot directly connect the output of one function to the input of another. To work around this language restriction, we leave it to the scope (the function `z` in this example) to handle 'wiring' functions to each other. Another issue is that C functions can have only one return value, but we can solve that problem by using custom return structs for each function. Below is the resultant C code from the conversion process.

```
struct f_output
{
  int r;
  int s;
};

struct h_output
{
  int t;
  int u;
};

struct g_output
{
  int v;
};

int z (int input)
{
  //The names of these variables do not matter.
  struct f_output f_out = f(input);
  struct h_output h_out = h(f_out.s);
  struct g_output g_out = g(f_out.r, h_out.t);
```

```
  int output = h_out.u + g_out.v;

  return output;
}
```

Note that the parallelization process expects the program to follow the above naming convention for the structs. Namely, the struct that holds the return value from a function `example` will need to have the name `example_output`. Graphical dataflow programs do not have named outputs or inputs, so the names of the struct elements can be internally generated.

This conversion works for any valid graphical dataflow function. If a graphical dataflow function cannot be serialized in this manner, it must have a circular dependency and, thus, be invalid. An example of a graphical dataflow function with a circular dependency can be seen in Figure 2.7.

2.2.3 Scopes

Due to an unfortunate implementation detail, scopes need to follow two conventions concerning variables passed in from the parent scope, and passed back to the parent scope. Section 3.2.3.1 contains the procedure that causes these conventions to be necessary[1]. If these conventions are not followed, it is possible that a scope will try to use a variable in the C representation that is not ready yet, which will cause unpredictable results.

These conventions are:

- Any variables in the C representation that the scope uses that are defined in a parent scope *must* be reassigned to new local variables before the scope is entered.

[1]Essentially, if an input to the scope or output from the scope is the return value from a function call, the procedure in Section 3.2.3.1 will not check `counter` before it is referenced again. The procedure only checks `counter` according to information local to the scope, and not with information from any subscopes or parent scopes. It may eventually be possible to improve upon this process to remove the need for the scope conventions.

- If a return value from a function is wired as an output of the scope, the return value must be assigned to a temporary variable before being assigned to the output.

For example, consider the following scope:

```
int b = getInt();
int output;

begin scope
  output = functionCall(a, b);
end scope
```

This would become the following:

```
int b = getInt();
int output;
int a_scope = a;
int b_scope = b;

begin scope
  int temp = functionCall();
  output = temp;
end scope
```

2.2.4 Loops

As mentioned previously, our graphical dataflow language is only concerned with two types of loops: `for` loops, and `foreach` loops. Both of these constructs define new scopes, and will have to follow scope conventions as explained in Section 2.2.3 in addition to the conventions below.

2.2.4.1 `for` Loop

Much like function calls, `for` loops are best explained with an example. Below is the C representation of the graphical dataflow function in Figure 2.9:

```
//Prerequisite: Pointer to input array is stored in variable input.
//Output array is allocated to the minimum of the input array's
//size and the value of N, and a pointer pointing to it is stored
//in variable output.
```

```
//Iterator
int i;

//Recirculating value
int recirc = 0;

//Addition accumulator
int add = 0;

//Stop condition
bool stop = false;

for (i = 0; i < input.size && i < 10 && !stop; i++)
{
  int tmp = input.array[i] + recirc;
  output.array[i] = tmp;
  add += tmp;
  recirc = tmp;
  stop = tmp > 7;
}
//Output may have been allocated larger than its current size;
//update the size to the number of elements that we have
//placed into it.
output.size = i;
```

The conversion is rather straightforward. Since the plugin makes no attempt to parallelize `for` loops, the translation is lax. However, any function calls within the loop body must follow the conventions established in Section 2.2.2.3, since they will be executed as concurrent function invocations.

2.2.4.2 `foreach` Loop

Since `foreach` loop iterations can execute independently, they are massively parallel. There is no `foreach` loop in C, so we must make due with using the `for` loop.

To make it easy for the parallelization process to recognize `foreach` loops from `for` loops, the loop body must be converted into a separate function call. The return value for this function call will be a struct with elements for:

- Inputs to aggregators.

- Inputs to accumulators.

The rest of the body of the loop will take the return values from the function call and update the accumulators and aggregators as necessary.

Below is the appropriate C code for the `foreach` loop in Figure 2.10:

```
//The output struct for the loop body's function. This is needed to
//follow the function call specifications explained in a previous
//section.
struct loop_body_output
{
  //Contains the output from an iteration to add to the 'addition'
  //accumulator
  int add;
};

//This is the function that contains the code for the loop body,
struct loop_body_output loop_body(float element)
{
  //In this trivial case, we are literally just returning the input.
  struct loop_body_output output = {element};
  return output;
}

//Code segment below is within the scope that contains the loop.

//Prerequisite: Pointer to input array is stored in variable input.

//Iterator
int i;

//Addition accumulator
int add = 0;

for (i = 0; i < input.size; i++)
{
  struct loop_body_output output = loop_body(input.array[i]);
  //Update accumulator with the output from the loop body.
  add += output.add;
}
```

2.2.5 `main` Method

Since we are translating graphical dataflow programs into C, there needs to be a `main` method defined that kicks off the program. The only purpose of this `main` method is to process inputs, and pass them into the function that is being run.

CHAPTER III

Methodology

The automatic parallelization process proposed in this chapter has three main components. First, we define an activation record format that allows multiple functions to execute concurrently and grants the ability to suspend and resume function execution when needed using possibly different threads. Next, we discuss how to overcome synchronization issues related to function dependencies. Finally, we describe a runtime library that performs operations that are integral to the execution of a parallelized program.

The net result of these structures is the *concurrent function invocation*, which allows the calling function to continue executing until it requires the return value from that call. The parallelization process converts all user-defined function calls into concurrent function invocations. This is best illustrated with an example:

```
int sum(int a, int b){return a + b;}
int subtract(int a, int b){return b - a;}
int application()
{
  int a = sum(1, 3);
  int b = sum(3, 5);
  int c = subtract(a, b);
  return c;
}
```

When the program starts, `sum(1, 3)`, `sum(3, 5)`, and `subtract(a, b)` are launched in parallel. The two calls to `sum` proceed to execute, since their arguments are avail-

able. The call to `subtract` must wait, since its arguments depend on the results from the calls to `sum`. It will execute only when its arguments are available. Meanwhile, the function `application` will pause at the `return` statement, since it must wait for the return value from the call to `subtract`. Once that value is available, it will execute the `return` statement.

Library functions, such as `printf`, are launched as regular functions since the libraries are not compiled with our automatic parallelization process. We can explain this using another example:

```
int sum(int a, int b){return a + b;}
int subtract(int a, int b){return b - a;}
int application()
{
  int a = sum(1, 3);
  int b = sum(3, 5);
  printf("Hello!\n");
  int c = subtract(a, b);
  return c;
}
```

Like before, `sum(1, 3)`, `sum(3, 5)`, and `subtract(a, b)` are launched in parallel. The calls to `sum` execute, while the call to `subtract` must wait for its arguments. `printf("Hello!\n")` is launched as a regular function call, meaning it must complete before `application()` can continue executing. While the call to `printf` is executing, the call to `subtract` may run if it receives its arguments from the two calls to `sum`. However, the `return` statement cannot execute until `printf` returns, and also until `subtract` returns the variable `c`.

This is the general idea behind our automated parallelization process.

Return value
Return address
Location of Caller's Activation Record
Function Arguments
Local Variables

Figure 3.1: This diagram shows the typical structure of a GCC activation record. Activation records should be fairly similar in other compilers.

3.1 Activation Records

3.1.1 Traditional Activation Records

By way of background, a traditional activation record in most languages stores the information used by a single function call. When a function `a` is called, a new activation record is created. `a`'s activation record contains space where its return value will be stored, the return address where `a` should jump to when it completes, the location of the caller's activation record, the arguments passed into `a`, and space where `a` can store local variables. In most programs, this activation record is stored in a `stack frame` and is pushed onto the top of the call stack. Once the activation record is constructed, the function `a` begins executing. When `a` finishes executing, it pops its stack frame off of the stack, restores the caller's frame pointer, pushes its return value to the top of the stack, and jumps to the return address. The structure of a normal stack frame in GCC is displayed in Figure 3.1.

If `a` calls another function (e.g. function `b`), an activation record is created for `b`, pushed onto the call stack, and `b` begins executing in the same manner as `a`.

This works fine for single threaded applications, where only one function is executing at a time and calling functions are suspended until called functions complete. The program only needs to keep track of the location of the current stack frame and the top of the stack in order to access information in the activation record. However,

this does not work when multiple functions are executing in parallel; the program needs to keep track of the location of each executing function's activation record.

Most applications that take advantage of parallel processing use traditional threads to execute tasks in parallel. Each thread has its own stack that stores activation records, so this implementation is nearly equivalent to making each parallel task a separate single threaded application. Threads of this kind are usually managed by the operating system, which can make thread management slow as it requires calls into the bulky operating system. Our need for fine-grain parallelism forces us to take a different route that avoids using the stack whenever possible.

3.1.2 Our Activation Records

Our modified activation records had to solve a few different problems. First, we need to allow functions to be 'wired' to one another, such that a function can write its return value directly into the activation record of another function as an argument to that function. Next, we need to allow functions to save state and suspend execution when they are waiting on return values from other functions, and to resume when those values are available. Finally, we need to allow multiple functions to execute in parallel without using a stack. All of this needs to be done without introducing race conditions or other synchronization issues.

In our fine-grain execution model, we changed the content and the location of activation records to meet these needs. While activation records are normally stored on the stack as stack frames, we opted to store them on the heap, as in the Mesa programming language (7). Using a heap allows multiple functions to execute concurrently and finish out-of-order. As explained below, it also allows return values to be forwarded directly to other functions represented by other activation records.[1] These activation records still store the arguments passed into the function and local

[1] To the criticism that using the heap will impact performance, we note that efficient heap allocators are possible and do exist. One such example is the Linux slab allocator (8)(p194-201).

`counter` variable & `isRunning` flag
Function entry point
`return_locations` structs for each return value
Function Arguments
Space to save program counter
Space to save registers' states
Local Variables

Figure 3.2: This diagram shows the structure of our modified activation records that are stored on the heap. The `counter` variable and `isRunning` flag are stored as one machine word to allow atomic instructions to modify and/or view both in one instruction. See Section 3.1.5.2 for more information.

variables.[2] However, they also contain space to save register values and the program counter, an entry point for the function associated with the activation record, a `counter` variable, `isRunning` flag, and `return_locations` structs. All of these are stored at fixed offsets from the start of the activation record, which is important; if a function knows the address to an activation record, it can calculate the addresses of these items to manipulate them. The space to save register data and the program counter allows functions to suspend and resume. The `counter` variable, `isRunning` flag, and `return_locations` structs allow functions to be 'wired' to each other. The details behind these features are explained in later sections of this chapter. A diagram of our modified activation record's structure can be seen in Figure 3.2.

3.1.3 Allocating Activation Records

In a normal C program, an activation record is created at the time a function is called. However, in our implementation of our graphical dataflow languages, an activation record is created at the start of the *scope* that calls its function (see Section 2.1.4 for details on what constitutes a scope). This is necessary because any

[2]Our graphical dataflow language does *not* have a concept of 'local variables'. However, the C representation of graphical dataflow programs specified in Chapter 2 creates local variables that represent wires. These must be stored in an activation record.

function called within that scope may need to write a value into the activation record of another function in that scope or in a parent scope.

We can explain this requirement using the example scope in Figure 2.6. Inside `z`, `f`, `g`, and `h` require activation records. Since all of these are in the same scope, and there are no conditionals affecting whether or not certain functions will execute, we know that `z` will need to eventually allocate activation records for all 3. The question is: When should we allocate them? Before `f` is launched, `z` will need to create activation records for `h` and `g` since `f` needs to write argument values to them. And, since `f` is called immediately, all of the activation records will need to be allocated right away. We could perform this dependency analysis on all programs to determine the latest point at which an activation record can be allocated, but the benefit of doing this is minimal if we know that `z` will need to use all of these activation records eventually. Thus, we always allocate all activation records for a scope at the start of the scope.

However, what happens if we have a conditional switch? Surely, we do not want to create the activation records of the function calls within each case of a conditional switch if there is a chance that they will not be needed! As mentioned in Section 2.1.4, a conditional switch creates a new scope.[3] Therefore, function calls within a case of a conditional switch will be created when that case begins executing.

By following the above rules, every activation record that a program allocates will eventually be used, so no memory or time is wasted. This model also supports recursion; a function can call itself, as long as a conditional checks whether or not the function should execute (otherwise, the function would always call itself, which would always call itself, ad infinitum).

[3]It is important to understand that a *scope* is not the same as a *function*. Unlike a function call, a scope does not have its own activation record.

3.1.4 Instantiating Activation Records

Before a function can begin executing, its activation record needs to be completely instantiated. In other words, the function call's arguments, the locations to which the function needs to write its return values, and other information that describes the function's initial state need to be written into the activation record before its function can begin executing.

First, let's discuss the return value locations. In our runtime library, we have the following struct:

```
struct return_locations
{
  void* ar_ptr;
  unsigned int offset;
  struct return_locations* next;
};
```

This struct describes a linked list of locations to which copies of one return value need to be written. One linked list of these structs is defined for each return value of a function, and a pointer to the head of each linked list is placed at a fixed offset in the activation record. This pointer can be `null` if the corresponding return value is not used at all. A function is not aware of the context in which it is being called, so it is the responsibility of the parent scope to dynamically allocate these structs, and to write a pointer to the head of each linked list into the function call's activation record. It does this at the same time that it allocates the activation records. More details on how they are actually used by a function can be seen in section 3.2.4.

Let's look at a quick example to illustrate how a scope allocates and instantiates `return_locations` structs. In Figure 3.3, the scope for the function `x` needs to set up three activation records. The resultant `return_locations` structs can be seen in Figure 3.4. Note how we handle return values that go to operations (e.g. `+`), and to multiple locations (e.g. `f`'s return value `s`). Because operations are carried out by the parent scope (in this example, the parent scope is the function `z`), return values that

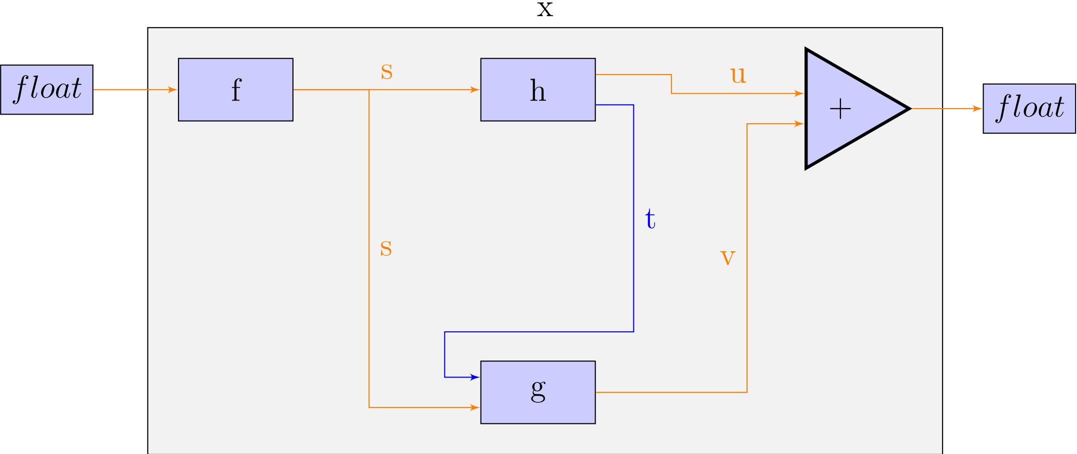


Figure 3.3: A sample graphical dataflow function. Note that `f`'s return value `s` needs to be written into the activation records for `h` and `g`.

are used in operations need to be written into the parent scope's activation record.

Arguments are more complex. A function is ready to run once it has all of its arguments. And, since arguments can come from multiple locations, we need a parallelizable way of launching the function immediately after the last argument is copied into the function call's activation record.

This is the first place where `counter` comes in. The `counter` in an activation record is initialized to the total number of arguments needed by a function. Each time an argument is copied into the activation record, `counter` is atomically decremented by 1. The activation record is considered to be *suspended* until all of the arguments are copied into the activation record and `counter` becomes 0, at which point the activation record is sent to the runtime library for execution. See Section 3.1.5 for more information on how a suspended function eventually executes.

The `isRunning` flag in an activation record indicates whether or not the activation record is running. We define *running* to mean that the activation record is ready to execute on the CPU or is currently executing on the CPU. This flag is used to prevent a race condition that would otherwise occur when activation records suspend as described in Section 3.1.5. `isRunning` must be initialized to 0, since the activation

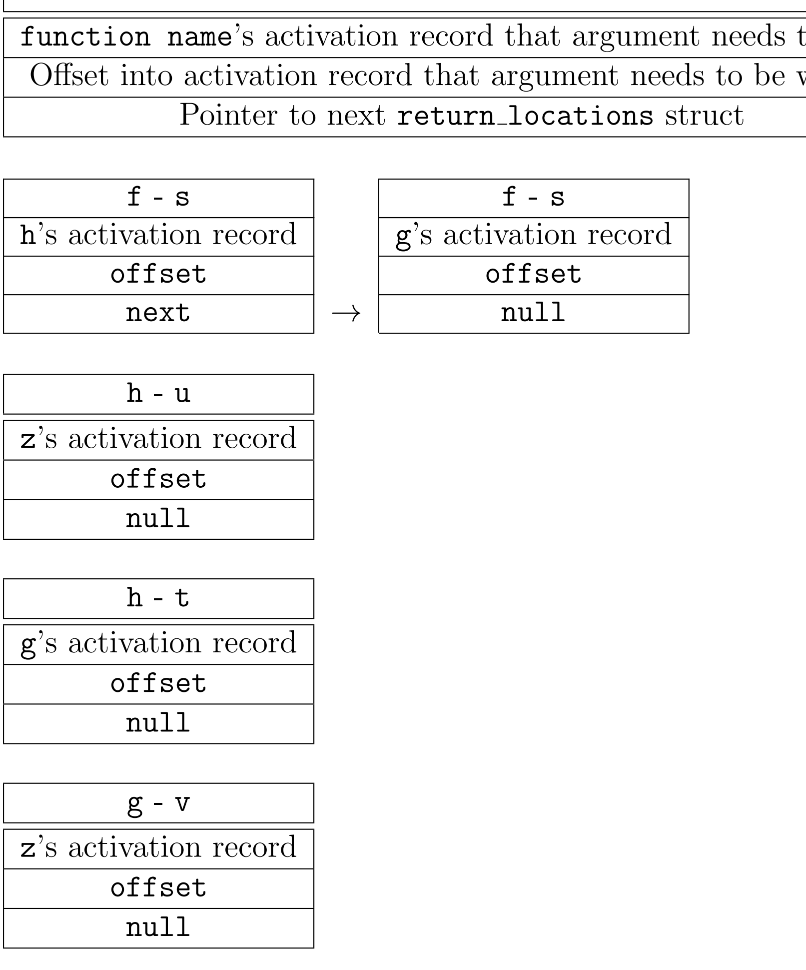


Figure 3.4: Example `return_locations` structs generated by the scope in Figure 3.3.

record does not have any of its arguments yet and is *not* ready to execute. In the event that a function does not require any arguments and is, in fact, ready to execute immediately after its activation record is allocated, then it does not matter what its `isRunning` flag is initialized to, since it will be set to 1 when the activation record is passed to the runtime library for execution.

As seen in Figure 3.2, both the `counter` variable and `isRunning` flag are stored in a single machine word. The reason for this decision is outlined in Section 3.1.5.2.

The *function entry point* in an activation record should be initialized to contain the address to the set of instructions that "resume" the activation record. In other words, the instructions should load the register states and program counter saved in

the activation record. An activation record will always begin or resume execution on the CPU by jumping to the function entry point. See Section 3.2.1 for details on how we implemented this.

The final pieces of information that need to be initialized in an activation record are the program counter and register states. In our implementation, every function begins in the *function entry point* described in the previous paragraph, so we must initialize the register states and the program counter to the same values that the function expects when it begins. In other words, all of the registers should be cleared (set to 0), and the program counter should be set to the address of the first instruction in the function's body.

3.1.5 Suspending and Resuming Activation Records

There are two scenarios in which an activation record must be suspended: it does not have all of its arguments, or its function needs to perform an operation using another function's return value that is not ready yet. In the first case, the activation record is initialized in a suspended state by its parent scope. In the second case, the function actually saves its own state and suspends execution.

Both of these cases use the `counter` to track when the function needs to resume, but in different ways. Let us describe each in detail, before generalizing how to deal with resuming suspended activation records.

3.1.5.1 Activation Records Waiting For Arguments

When an activation record is initialized, `counter` is set to the number of arguments that its function requires. These arguments can come from multiple different functions, which may be executing concurrently. Each time an argument becomes available, the function providing the argument must perform the following procedure:

1. Write the argument into the destination activation record at the desired offset.

2. Atomically decrement the `counter` of the destination activation record by 1. This operation returns the new value of `counter`.

3. If `counter` becomes 0, the function must resume the activation record by using the appropriate runtime library method (see Section 3.3.2 for more details). The important takeaway from this is that it is the responsibility of the function that supplies the final argument to an activation record to start the function associated with it.

A suspended activation record that is waiting for arguments can *only* resume when its counter decrements to 0. The atomic decrement operation prevents synchronization issues, race conditions, and misses. If we did not use an atomic decrement operation, then it would be possible for two (or more!) different functions to decrement the counter at the same time before checking its new value. If the second function decrements it to 0, then both would see the counter at 0, and the function would execute twice! Since the atomic decrement operation is hardware supported, it is a fast and computationally cheap method to ensure that this does not occur.

Notice that this operation requires the function providing the argument to have two key pieces of information: the address of the destination activation record, and the offset at which the argument needs to be written. If the argument is supplied as the return value of a function call, then the address of the destination activation record and the appropriate offset are recorded in the `return_locations` structure for that return value. In all other cases, the argument is supplied by an operation performed in the parent scope that created the destination activation record in the first place. So, it should have a reference to the destination activation record and the appropriate offset that the argument needs to be written at.

3.1.5.2 Activation Records Waiting For Return Values

A scope may use return values from its function calls in operations. However, because the function that the scope belongs to and its function calls execute concurrently, we need a way to be sure that a return value from a function call is ready before its scope tries to use it. Here, we must use both the `counter` variable and `isRunning` flag to ensure that the function can suspend when a needed return value is not available and resume when the needed return value is available with no race conditions.

The value of `counter` in this scenario is the number of obtainable return values that the function will need to use in its own execution. At the start of each scope, and before writing certain values to other activation records as arguments, the function must add a constant determined at compile time to its own `counter`. See Section 3.2.3 for specific details on how these constants are determined. Eventually, function calls within the scopes contained within the function will decrement `counter` to 0, as described in a procedure later on in this section.

If a function gets to the point of needing a return value before `counter` reaches 0, it must suspend itself. As a result, a function must check its `counter` before trying to use the return value in an operation. If `counter` is not 0, then it must prepare to suspend itself by saving its registers and program counter! It must also re-check `counter` after preparing to suspend to ensure that the return value did not become available while it was preparing to suspend. In addition, it must change its `isRunning` flag to 0 at the same time to prevent a race condition where the return value is supplied in-between these two operations. Atomic instructions only operate on one machine word at a time, so we decided to store both of these values in one machine word, where the lowest bit is `isRunning`. So, for a machine that operates on words of size n, this word would have the following layout:

Bit:	$n-1$	...	1	0
Contents:	`counter`			`isRunning`

With this in mind, a function can successfully suspend itself by performing the following procedure:

1. Save a copy of the current value of the `counter` and `isRunning` word.
2. If `counter` is greater than 0:
   (a) Save the registers that have been used into the activation record.
   (b) Save the address of the first instruction in the operation that uses the return value into the program counter field in the activation record so the function resumes at that spot when the return value becomes available.
   (c) Try to perform a compare-and-swap on the `counter` and `isRunning` word, where the old value is the saved copy of the `counter` and `isRunning` word, and the new value is the old value minus 1 (which sets the `isRunning` flag to 0).
   (d) If the compare-and-swap fails:
      i. Update the saved value of the `counter` and `isRunning` word to the current value of the `counter` and `isRunning` word.
      ii. If `counter` is now 0, jump to the operation that uses the return value, since the return value is now available and the activation record no longer needs to suspend.
      iii. Otherwise, jump to the compare-and-swap operation above to repeat this procedure.
   (e) If the compare-and-swap succeeds, call the appropriate runtime library function to complete the suspension of the activation record.

3. Else, `counter` must be 0, so the return value must be available for use in an operation. Continue executing.

From the other end, each time a function writes a return value into an activation record for use in an operation, it must follow a procedure similar to what was described for arguments. The main difference is that the activation record it is modifying may be running when its `counter` is decremented to 0, in which case it does not need to be resumed. With this in mind, the function providing the return value must perform the following procedure:

1. Write the argument into the destination activation record at the desired offset.

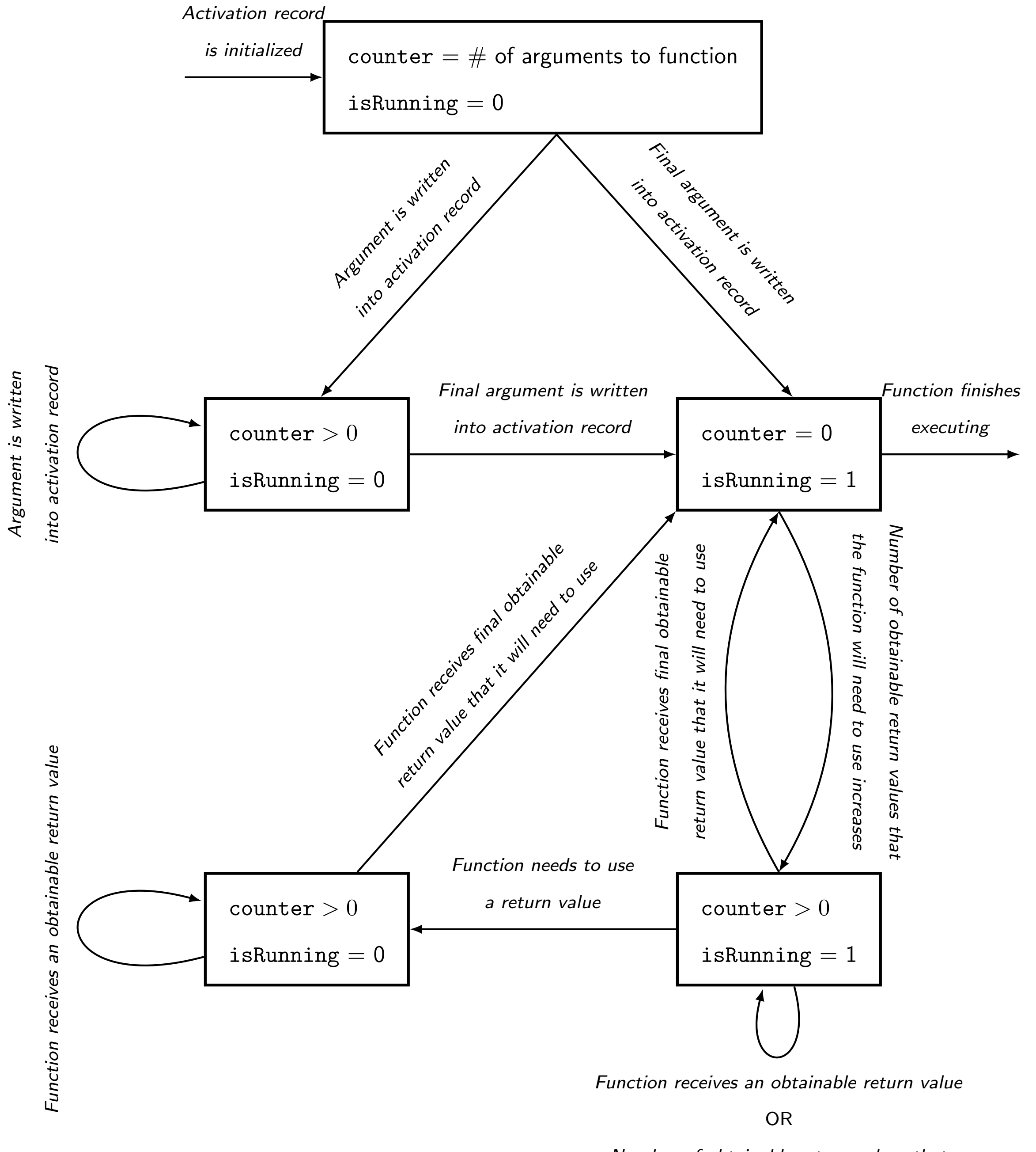
2. Atomically decrement the `counter` and `isRunning` word of the destination activation record by 2. This effectively decrements the `counter` portion of the word by 1, and will return the new values of `counter` and `isRunning` in one word.

3. If `counter` and `isRunning` are both 0 (which would make the entire `counter` and `isRunning` word 0), the function must resume the activation record by using the appropriate runtime library method (see Section 3.3.2 for more details).

3.1.5.3 Generic Procedure for Resuming Activation Records

As mentioned in Section 3.1.4 on initializing activation records, `isRunning` is initialized to 0, and remains that way until the activation record resumes. So, whenever an argument is written into an activation record, `isRunning` is 0. This allows us to use the procedure for writing return values into activation records for operations for all cases when a value is written into an activation record.

The following state diagram and table summarize how `counter` and `isRunning` are managed throughout the life of an activation record:


Activation record
is initialized
counter = # of arguments to function
isRunning = 0
Argument is written
into activation record
Final argument is written
into activation record
Argument is written
into activation record
counter > 0
isRunning = 0
Final argument is written
into activation record
counter = 0
isRunning = 1
Function finishes
executing
Function receives final obtainable
return value that it will need to use
Function receives final obtainable
return value that it will need to use
Number of obtainable return values that
the function will need to use increases
Function receives an obtainable return value
counter > 0
isRunning = 0
Function needs to use
a return value
counter > 0
isRunning = 1
Function receives an obtainable return value
OR
Number of obtainable return values that
the function will need to use increases

Event	**Action to `counter`**	**Action to `isRunning`**
Activation record is initialized.	`counter` is set to the number of arguments in the function.	`isRunning` is set to 0
Argument is written into activation record.	`counter` is decremented by 1.	If `counter` is now 0, `isRunning` is set to 1.
Function fulfills final obligation to a function call by either providing it with the last argument it is responsible for, or by launching it.	`counter` is incremented by the number of new "obtainable" return values resulting from this function call (see Section 3.1.5.2).	Nothing.
Function requires the use of a return value.	None.	If `counter` $>$ 0, `isRunning` $=$ 0 and the activation record suspends. Otherwise, nothing happens.
Function is running, and receives a return value from a function call.	`counter` is decremented by 1.	Nothing.
Function is suspended, and receives a return value from a function call.	`counter` is decremented by 1.	If `counter` is now 0, `isRunning` $=$ 1 and the function resumes.

3.2 Program Transformations

Now that we have explained our modified activation record format and how it is used, we need to modify the program code to take advantage of it. The following section describes these transformations in detail.

3.2.1 Function Entry Point

As described earlier, the function entry point of a function is responsible for loading the saved registers and program counter from the function's activation record in order to resume the function's execution on the CPU. Since each function may use

different sets of registers, we decided to only save the registers that a function actually uses in its activation record. Because of this, the space in the activation record used to save registers will be laid out differently depending on the subset of registers that the function uses. And, due to this, a unqiue function entry point procedure must be defined for each subset of registers used by functions in the program.

To make implementation simpler, we opted to define a unique function entry point for every function. We added instructions to the start of every function to load the registers and program counter saved into its activation record. Due to this procedure, the function entry point for a given function is just a pointer to the start of the function. One implication of this is that we need to define a label for the start of the function body of each function, since a function's scope must initialize its saved program counter to the start of the function body.

After this transformation, a function will look like the following pseudocode:

```
begin function example
  load registers
  load program counter
  label example_function_body
  //Insert code for function body
end function example
```

Register information is not available until late in the compilation, so it may be necessary to insert a placeholder for the register loading code until register information is available.

3.2.2 Establish Function Argument Order

The next important step that the optimization pass needs to perform is to establish some method of ordering the arguments to a function. While arguments have a clear order in C, compilers may shift things around. For our optimization pass, the order of arguments is very important, because the optimization pass needs to calculate the

offset of each argument of a function with respect to the function's activation record.

One method for ordering arguments is to sort them by name, if the original function parameter names are available. The arguments to a C function must have unique names, so there is no ambiguity provided that the same order is used throughout the optimization pass.

3.2.3 Entering a Scope

Whenever a scope is entered, the following three actions need to occur in the following order:

1. The scope will need to allocate and initialize the activation record of each function call within that scope.
2. The scope may need to add a constant value to `counter`.
3. The scope will launch any of the activation records that it has initialized that already have a `counter` of value 0. This will occur if one of its function calls requires no arguments.

We will now explain the program transformations required for each of these procedures.

3.2.3.1 Allocating and Initializing Activation Records

In Sections 3.1.3 and 3.1.4, we explained how activation records are allocated and initialized, but not how the compiler achieves these results. At the start of a new scope, the optimization pass needs to collect the following information in order to properly initialize activation records:

- Obtain a list of all of the function calls made inside the scope.

- Detect the source of each argument of each function call. Arguments can either be supplied by the scope itself, or by a return value from a function call.

- Detect the destinations of each return value of each function call. Return values can be used by the scope and in other function calls as an argument. Remember that each return value can have multiple destinations!

Note that the last two items in the list are very much related. The destinations of many return values will be arguments to function calls. Any argument that is *not* supplied by a return value *must* be supplied by the scope. We use this result in the optimization pass.

We will describe how to collect this information as a series of algorithms, since the procedures involved are quite complex. All of the algorithms occur *in the compiler* as an optimization pass. So, when an algorithm defines a structure, it means *in the optimization pass*, and not in the graphical dataflow program that the compiler is compiling. When the algorithm mentions *inserting* something, it means to insert that into the graphical dataflow program being compiled. Futhermore, our algorithms operate on an abstract syntax tree (AST) representation of the program. We assume that this AST representation has a node type for 'scope' that represents a new scope.

For a first pass through the scope, the optimization pass collects the information it needs to initialize the activation records for the function calls in its scope, and performs program transformations that can be made along the way with the information it collects, such as inserting the code to wait for return values.

```
//This is used to track the destinations of each return value.
//Note that this is different from return_locations! return_locations
//is a structure that exists at runtime. This structure exists during the
//compiler optimization pass.
define a new structure called return_destinations that contains:
  a reference to one return value node
  a list called destinations of variable and function argument nodes

initialize function_calls, a list of function call nodes
initialize return_destinations_list, a list of return_destinations
```

```
for each child node of the scope body:
  //This child scope will be processed later using this procedure.
  if node is a scope node:
    skip over scope node
  //Note that calls to library functions are ignored and unmodified.
  else if node is a function call to a user defined function:
    add function call node to function_calls
    for each return value:
      create a return_destinations structure
      associate the return_destinations structure with the return value
      add return_destinations to return_destinations_list
      for each argument:
        if source of argument is a return value in return_destinations_list:
          add argument to destinations in the return value's return_destinations structure
  else if node references a return value in return_destinations_list:
    //See Section 3.2.5 for details on this code.
    insert code just before this node to check the counter
    //The return values will no longer exist when we remove the function
    //call, so we need a new local variable that the function will write to
    //when the return value is available.
    replace reference to return value with a new local variable
    add new local variable to the list of destinations for the return value
```

This information is enough to enable the compiler to generate the code to allocate the activation records at run time, and to initialize them properly, by following this procedure:

```
//This structure only exists to associate function calls with
//their activation records.
define a new structure called ar_association that contains:
  a reference to the address of an activation record
  a reference to a function call node

initialize ar_associations, a list of ar_association structures

go to the start of the scope
//Allocate and instantiate most parts of each activation record
for each function call in function_calls:
  insert a new local variable of type void*
  //Note that this call to the runtime library will need the size of the activation
  //record as an argument. However, this value cannot be calculated until late in the
  //compilation. We can insert a symbol reference for this value at this point of the
  //compilation with the name FUNCTIONNAME_ARSIZE. The algorithm in Section 3.2.9.3
  //will calculate and set the values of these symbols.
  insert runtime library call to allocate the activation record for the function call,
    and to store result of call into new local variable
  add local variable to a new ar_association structure
  add function call to the ar_association structure
  //Since counter and isRunning are one machine word,
  //these two steps can be made in one instruction by
```

```
  //initializing the word to 2*number of arguments
  insert code to instantiate counter to the number of arguments to the function call
  insert code to initialize isRunning to 0
  insert code to initialize function entry point to the address of the start of the function
  //This address should be obtainable using the label created when the
  //function entry point was created.
  insert code to initialize program counter to the start of the body of the function

//Instantiate the return_locations structures for each return value of each activation record
for each ar_association in ar_associations:
  for each return value of the function call in the ar_association:
    insert a return_locations pointer to act as the head of the return_locations struct
    insert code to initialize the local variable to NULL
    for each destination in its return_destinations structure:
      insert code to dynamically allocate a return_locations struct
      insert code to initialize the next pointer in the struct to the head
      insert code to initialize the ar_ptr pointer to the activation record that
        contains the destination
      if the activation record is the scope that the function call belongs to:
        //Destination is a local variable in the scope, so its address will later be
        //determined by the compiler.
        insert code to calculate the offset by subtracting ar_ptr from the address
          of the destination
      else:
        //Otherwise, it is an argument to a function, which is at a location
        //that we define -- it's in the arguments section of the function
        //call's activation record!
        //This offset can be calculated using the argument ordering established
        //elsewhere, along with the sizeof operator.
        calculate the offset of the argument in the destination activation record
        insert code to set the offset of the struct to this constant
      insert code to make this return_locations struct the new head
    insert code to initialize the return_locations pointer for this variable in
      the activation record to the address stored in the head.
```

At the end of all of this, the scope will have code at the start to allocate and initialize its activation records. The information generated by this entire procedure will be used again in Section 3.2.4, where we describe how to insert the code to write arguments into activation records.

3.2.3.2 Adding a Constant to `counter`

We established in Section 3.1.5.2 that the value of `counter` takes on a new meaning after an activation record has all of its arguments. Specifically, it becomes the number of obtainable return values that the function will need to use during execu-

tion. 'Obtainable' means that these return values are supplied by the closed set of function calls that are able to execute without any direct or indirect dependencies on the current function. This closed set expands during runtime as the function resolves its dependencies with function calls by writing its return values into activation records, and by launching activation records at the start of scopes that do not require any arguments. In this section, we will discuss the latter case. For the former case, see Section 3.2.4.

The value of `counter` should be managed according to the algorithm below. We make use of a list of concurrent function invocations that we call `executingFunctions`. This list contains the previously described closed set of function calls. Note that this list corresponds to a *function*, and not just a *scope*! This list only exists during the algorithm, and is *not* an entity during runtime.

For each scope in the function [including the function's main scope]:

1. Add all of the concurrent function invocations in the current scope that do not require any arguments to `executingFunctions`.

2. Add any concurrent function invocations in the current scope and parent scopes that only receive arguments from functions in `executingFunctions` to `executingFunctions`, since these functions will eventually execute with no further input from the current function. Note that this only applies to concurrent function invocations not already in `executingFunctions`. Repeat this step until no new concurrent function invocations are added to `executingFunctions`.

3. As the first instruction in the scope, atomically add the total number of return values that the *entire function* uses from all of the concurrent function invocations just added to `executingFunctions` to `counter`.

3.2.3.3 Launching Activation Records that Require No Arguments

The final program transformation that occurs at the start of a scope is launching activation records that require no arguments. It is perfectly legal to define a function in our graphical dataflow language that takes no arguments. These functions will be able to launch as soon as their activation records are allocated. So, after the code that adds a constant amount to `counter`, the optimization pass will add calls to the runtime library to execute any activation records that it allocated that have a `counter` of 0.

3.2.4 Writing Arguments into Activation Records

Using the information from the algorithms described in Section 3.2.3.1, we can insert code to write the function call arguments that the scope is responsible for providing into the appropriate activation records. We also have to delete the function call nodes since, in our graphical dataflow language, functions *only* execute when they have all of their arguments!

We will describe this process as an algorithm. As with the algorithms defined in Section 3.2.3.1, this algorithm is executing inside the compiler as an optimization pass during the compilation of a graphical dataflow program:

```
for each child node of the scope body:
  if node is a scope node:
    skip through statements until subscope ends
  else if statement is a function call:
    //See below for this procedure
    calculate the constant counter increase that will be caused by this
    insert code to atomically increment counter by this constant
    for each argument that is not supplied by a return value:
      //See below for this procedure.
      calculate the address of the argument in the activation record
      insert code to write argument into function call's activation record
      insert code to atomically decrement the function call's activation record's
        counter and isRunning word by 2
      insert code to launch the activation record if its counter and isRunning word is 0
    remove function call from scope body
```

This algorithm leaves two subprocedures undefined: how to determine the constant to increment `counter` by, and how to calculate the address of an argument in an activation record.

A similar method to what was described in Section 3.2.3.2 can be used to find the constant to increment `counter` by. It reuses the `executingFunctions` list that is defined in the compiler for the function that the scope occurs in:

1. Add the function to `executingFunctions`.
2. Add any concurrent function invocations in the current scope and parent scopes that only receive arguments from functions in `executingFunctions` to `executingFunctions`, since these functions will eventually execute with no further input from the current function. Note that this only applies to concurrent function invocations not already in `executingFunctions`. Repeat this step until no new concurrent function invocations are added to `executingFunctions`.
3. As the first instruction in the scope, atomically add the total number of return values that the *entire function* uses from all of the concurrent function invocations just added to `executingFunctions` to `counter`.

For calculating the offset of an argument in a function call's activation record, the algorithm can simply add the following together:

$$\begin{aligned}\text{Offset} = {} & \text{Address of Activation Record} + \text{Offset where } \texttt{return_locations} \text{ structs begin} \\ & + \text{\# of return values} * \text{size of a pointer} + \text{size of } \texttt{counter} \text{ and } \texttt{isRunning} \text{ word} \\ & + \text{sum of the size of each argument that occurs before this argument in} \\ & \quad \text{the argument ordering (see Section 3.2.2)}\end{aligned}$$

All of these numbers except the address of the activation record are available at

some point during the compilation process, so it is possible to calculate and insert a constant offset into the program rather than have the program calculate it at runtime. It would simplify the runtime calculation to:

$$\text{Offset} = \text{Address of Activation Record} + \text{Constant Offset}$$

As a side note, we could optimize the argument writing procedure to write the arguments into an activation record once they are available, rather than at the location where the function call occurs in the C representation. Namely, the optimization pass could find the latest point in the scope where one of the arguments to the function call was modified, and write the arguments into the activation record there. However, this adds some issues to our procedure for determining the constant to increment `counter` by. Namely, look at the following code:

```
begin scope
  int a = 2;
  int b = firstFunc();
  int c = b+1;
  int d = secondFunc(c);
  int e = thirdFunc(a, d);
  int f = d+1;
end scope
```

Going strictly by the optimization, we could insert the code to increment `counter` and write `a` into `thirdFunc`'s activation record directly after `int a = 2`. `counter` should be increased by 1, since the return value `e` is used in an operation. Now, the code looks like this:

```
begin scope
  //counter is initialized to 1, due to firstFunc()
  counter = 1
  runtime library call to start firstFunc
  int a = 2;
  increment counter by 1
  //counter may be 2 if firstFunc() has not returned yet.
  write a into thirdFunc's activation record
  atomically decrement thirdFunc's counter
  //thirdFunc cannot possibly have a counter of 0, since it relies on the return value
  //from secondFunc which receives an argument from an operation that occurs later on
```

```
  //in this scope.
  launch thirdFunc if its counter is 0 and isRunning is 0
  //b is a return value from firstFunc, so we need to make sure it is available before
  //using it (e.g. check that counter is 0). Counter is at most 2 at this point. However,
  //it will NEVER decrement to 0; it expects firstFunc to decrement it once, but it cannot
  //because it is waiting for an argument from secondFunc, which is waiting for this
  //operation to complete! We have deadlock!
  if counter > 0
    suspend //Represents entire suspension process
  int c = b+1;
  increment counter by 1
  write c into secondFunc's activation record
  atomically decrement secondFunc's counter
  launch secondFunc if it has a counter of 0 and isRunning is 0
  if counter > 0
    suspend //Represents entire suspension process
  int f = e+1;
end scope
```

There are probably ways to get around these deadlock situations (e.g. changing where `counter` is incremented)[4], but they are outside the scope of this project.

3.2.5 Waiting For Return Values

In Section 3.2.3.1, we determined where to place the procedure to wait for a return value from a function. In this section, we will explain how this procedure works.

First off, this procedure does not have to run every time we use a return value. If the function has checked `counter` previously, and `counter` has not been incremented, then `counter` must be 0. Thus, this procedure only needs to execute if `counter` has been incremented since the last `counter` check.

Section 3.1.5.2 gives an excellent overview of the procedure underlying this program transformation. It translates nicely into the following pseudocode that should be inserted by the optimization pass in the compiler just before a return value is used (provided that `counter` has been incremented since the last `counter` check):

[4]This position needs to be selected carefully. `counter` is unsigned, so it needs to be incremented in a place that is guaranteed to execute before the function that is going to decrement it, otherwise a function could try to decrement it when it is 0. But, it cannot execute too soon, or else the deadlock situation will occur.

```
unsigned int savedCounterWord = current value of the counter and isRunning word in the AR
//The counter and isRunning word will be at least 1, since isRunning is set to 1.
if savedCounterWord > 1:
  //See note below about register saving.
  save the registers used by the function into the activation record
  //There should be a node in the compiler that represents the address of the
  //instruction that a node represents. Use this to retrieve the address of the
  //node that represents the instruction that uses the return value.
  save the address of the operation that uses the return value into the program counter
    field of the activation record
  //compare_and_swap returns a boolean.
  //Repeatedly try to flip the isRunning bit to 0.
  while !compare_and_swap(address of counter and isRunning word,
    savedCounterWord, savedCounterWord-1):
    savedCounterWord = current value of the counter and isRunning word in the AR
    //A value of 1 indicates a counter of 0 with isRunning set to 1.
    if savedCounterWord == 1:
      //Jump to the instruction that uses the return value since the counter is 0.
      //We saved this address into the program counter field.
      jump to the address in the program counter field of the activation record
  call the runtime library function to suspend the activation record
```

One part of that algorithm that needs clarification is the process to save the registers used by the function into the activation record. In most compilers, it is not possible to do this at the abstract syntax tree (AST) level; registers are assigned near the end of compilation. Thus, the optimization pass should insert a dummy node at this part of the compilation that can be identified later on in the compilation process when registers are assigned and replaced with actual code.

3.2.6 `for` Loops

The body of a `for` loop is treated as a new scope. Meaning, at the top of the loop body, the activation records for each function call used in the loop body will be allocated and initialized. The function calls made in the loop body will be converted to concurrent function invocations. `for` loops are not massively parallel like `foreach` loops; only one iteration of the loop can execute at once due to recirculating values and stop conditions causing dependencies between loop iterations.

However, there is plenty of existing research concerning how to automatically parallelize `for` loops, so it is possible that a future implementation of our graphical

dataflow language will attempt to parallelize `for` loops (9).

3.2.7 `foreach` Loops

There are many bodies of work that explain how to parallelize `foreach` loops. In particular, the Fortress programming language has implicitly parallel `foreach` loops (6). These methods could be applied to our project, but applying them is outside of the scope of our implementation.

3.2.8 Function Termination

When a function ceases executing, it calls a runtime library method that returns its thread to the threadpool. More information about this can be read in Section 3.3.4.

3.2.9 Late Compilation Changes

A number of details, mentioned in previous sections of this chapter, cannot be resolved until later on in the compilation process. This subsection deals with these few loose ends.

3.2.9.1 Register State Saving and Loading

When suspending and resuming activation records, the program needs to save and load its register states, respectively. Each function uses some subset of the registers available on the CPU, so we have decided that the activation record of each function will only save the registers used by the function.

One important detail that needs to be sorted out is the order of the saved registers in each activation record. We assume that there is an unambiguous way to sort the registers on the CPU.

In previous algorithms that operated on the abstract syntax tree representation of the graphical dataflow program, we inserted dummy nodes for register loading and

saving procedures. We will be referencing them in the following algorithm, which describes how to insert the code to save and load registers. Note that this procedure takes place inside the compiler as an optimization pass:

```
for each function in the program:
  initialize func_register_list, a list of registers
  //We do not know what internal representation compilers typically use at this stage
  //(GCC uses RTL), so we use the generic term 'statement' to mean the representation
  //of an instruction.
  for each statement:
    if the statement references a register that is not in func_register_list:
      add the register to func_register_list
  sort func_register_list using the unambiguous sorting method that we assume exists
  //Now, we will look for the 'dummy nodes' inserted previously into the AST. They should
  //have been constructed in such a manner that they will be recognizable at this late stage
  //of compilation.
  for each statement:
    if the statement is a dummy node representing register loading:
      calculate the offset in the activation record where saved registers appear
      insert code to add the address of the activation record to this offset
      insert code to save the result of the operation into a new local variable called
        register_address
      //This list will already be sorted in the same order that they are saved into the
      //activation record.
      for each register in func_register_list:
        load the value saved at the address in register_address
        add the size of the register to register_address
      remove the dummy node statement
    else if the statement is a dummy node representing register saving:
      calculate the offset in the activation record where saved registers appear
      insert code to add the address of the activation record to this offset
      insert code to save the result of the operation into a new local variable called
        register_address
      //This list will already be sorted in the same order that they are saved into the
      //activation record.
      for each register in func_register_list:
        save the value in the register into the address contained in register_address
        add the size of the register to register_address
      remove the dummy node statement
```

3.2.9.2 Changing Argument Source

One common optimization that compilers perform is passing arguments in registers whenever possible. Unfortunately, this technique is impractical, since we write every argument into the activation record of a function call. We need to change the low level code to use the arguments that are stored in the activation record rather than

registers.

An easy way to do this is to insert code at the start of a function body to load arguments into the registers that the function expects them to be in. There needs to be some metadata available in the compiler that specifies this information. This process allows us to modify the program with minimal changes at the register level.

3.2.9.3 Calculating AR Size

In order to tell the runtime library to allocate an activation record, we need to know the size of the activation record of each function. This information can only be calculated late in the compilation process. In Section 3.2.3.1, we inserted a reference to a symbol with the name `FUNCTIONNAME_ARSIZE` when we needed to use this value. Now, we will define the value behind these symbols.

At a late phase of compilation, the compiler will determine the stack size of each function. This value represents the size of the part of the activation record that stores local variables. We can use this, along with other information we know, to calculate the total size of the activation record:

$$\begin{aligned}\text{Activation Record Size} = &\text{ size of } \texttt{counter} \text{ and } \texttt{isRunning} \text{ machine word} \\ &+ \text{size of a pointer} + \# \text{ of return values} * \text{size of a pointer} \\ &+ \text{size of program counter} \\ &+ \text{combined size of all of the registers that the function uses} \\ &+ \text{combined size of all of the arguments} + \text{stack size of function}\end{aligned}$$

This value needs to be calculated for each individual function, and stored into a symbol named `FUNCTIONNAME_ARSIZE`.

3.3 Runtime Library

The main duties of the runtime library are to allocate space for the activation records, manage a ready queue for activation records that are ready to execute, and to manage a threadpool that operates on the ready queue. Some of the functionality of the runtime library could be implemented in hardware in the future, but this section describes their implementation in software (1).

3.3.1 Allocating Activation Records

The runtime library provides a function that allocates an activation record. The size of the activation record of a function is known toward the end of the compilation, and must be exported as a symbol in the executable so it can be accessed when needed. In any case, the runtime library is provided with the required size of the activation record, and allocates it on the heap. It returns a pointer to the newly allocated activation record.

3.3.2 Ready Queue

The *ready queue* is a simple queue of activation records that are ready to execute on the CPU, but are waiting for a thread.[5] Access to the ready queue must be synchronized somehow to avoid race conditions. The design of the ready queue is beyond the scope of this MQP; the current design immediately runs every activation record that is ready to execute.

3.3.3 Resuming Activation Records

When an activation record's `counter` hits 0, it is ready to resume execution on the CPU. The runtime library should have a method for resuming activation records

[5] While it is called the ready *queue*, the activation records have no required order of execution, so an unordered data structure would work as well.

that does the following:

1. Sets the `isRunning` flag of the activation record to 1.

2. Adds the activation record to the ready queue.

3.3.4 Thread Pool

The runtime library manages a threadpool. Each thread will take an activation record from the ready queue and execute it on the CPU. When a thread finishes executing a function, it deallocates its activation record, and returns its thread to the threadpool, where it may be assigned to execute another function. If a function suspends (see Section 3.1.5 for details), it calls a runtime library method that returns the thread to the threadpool without deallocating the activation record of the function.

The threads in the threadpool only need to be created once, and can be reused for the duration of the program. Many thread pool implementations prune threads from the pool if the demand for threads decreases, and add additional threads if the demand increases. Using a thread pool greatly decreases the number of times that we run the expensive thread creation process, which requires a trip through the operating system. In the future, we hope to develop lightweight threads that do not use a stack, and are created and managed in userspace to avoid expensive operating system calls.

CHAPTER IV

GCC Background

Before delving into implementation specifics, we must explain a few details about the internal workings of GCC. The GNU Compiler Collection, or GCC, is a compilation system that evolved from the GNU C Compiler. Because it supports a wide number of languages, GCC's core has to compile and optimize programs in a language independent manner. Figure 4.1 shows the main phases of the GCC compiler. Let's discuss these phases in detail.

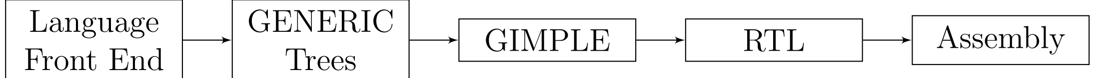


Figure 4.1: The GCC compilation pipeline.

4.1 Language Front End

GCC language front ends take care of language-specific details, such as lexical analysis and parsing. Language front-ends produce a GENERIC tree representation of the program, which is passed off to GCC's core. The GCC plugin hooks into the C language front end to save the GENERIC tree nodes that represent functions and `struct` definitions in the runtime library from the garbage collector.

4.2 GENERIC Trees

GENERIC trees are GCC's take on abstract syntax trees. This program representation exists primarily to provide a language-independent representation of an entire function. Each function is represented as a single GENERIC tree structure. GCC also allows languages to extend GENERIC trees with language-specific tree nodes, but this project did not need to deal with this additional feature because C programs only use the core GENERIC tree nodes.

After receiving the GENERIC tree representation of a program, GCC combs through it and garbage collects tree nodes that it is certain will not be needed for compilation. For example, it will garbage collect nodes that represent library functions and structs if they are not referenced by any function. So, if a program includes `stdlib.h` and only uses the `atoi` function, GCC will garbage collect the tree nodes that represent all of the other functions in `stdlib.h` to save memory. At this stage of the program, our compiler plugin will prevent the tree nodes that represent functions and `struct` declarations from the runtime library from being garbage collected.

When this process is complete, GCC does not perform any further optimizations on the GENERIC tree representation of the program. Instead, it converts the program into GIMPLE tuples (through a process that GCC calls *gimplification*), and proceeds to the next phase of compilation.

4.3 GIMPLE

GIMPLE is a tuple-based high-level internal representation language based upon and named after the SIMPLE intermediate representation used in the McCat compiler at McGill University (10). While the GENERIC tree structure exists to provide a language-independent way to represent functions, GIMPLE exists to provide a more restrictive internal program representation that is suitable for optimization passes.

There are GIMPLE tuple equivalents for most C statements, such as function calls, variable assignments, conditionals, return statements, etc. Most of these tuples have three or fewer elements (with function calls being a notable exception, since they require an element for each function argument). A list of these tuples can be seen in the GIMPLE chapter of the GCC Internals manual (11).

This phase of the compiler is broken up into three stages: High GIMPLE, Low GIMPLE, and SSA GIMPLE. Each of these forms is more restrictive than the last, as the GIMPLE representation will eventually need to be translated into the low-level RTL representation.

4.3.1 High GIMPLE

GENERIC trees are converted to a form of GIMPLE called *High GIMPLE*. Compared with other GIMPLE representations, High GIMPLE contains tuples for explicit lexical scopes and exception handling. Any optimizations that operate on these high-level structures will run on this representation.

At this stage of compilation, the compiler plugin will perform most of the program transformations described in Chapter III, including scope transformations.

Eventually, the GIMPLE tuples that are exclusive to the High GIMPLE representation are converted to simpler, but semantically equivalent, tuples. The result of this transformation is Low GIMPLE.

4.3.2 Low GIMPLE

Low GIMPLE is the product of linearizing (or lowering/inlining) the lexical scopes and exception handling tuples from High GIMPLE. Other than these changes, the representation remains the same.

4.3.3 SSA GIMPLE

SSA GIMPLE, or **S**tatic **S**ingle **A**ssignment GIMPLE, is a form of GIMPLE where every variable is assigned exactly once. GCC transforms the GIMPLE representation of a variable reference to include a 'version' number of the variable. Every time a variable is reassigned, the version number increases by 1, and all further references to the variable refer to the new version. Optimizations that run on this subset of GIMPLE typically eliminate unnecessary variables and variable assignments, e.g.:

```
int someFunction()
{
  //This assignment is pointless.
  int a = 3;
  a = 4;
  int b = 3;
  //The variable c is never used.
  int c = 2;
  return sum(a, b);
}
```

...becomes:

```
int someFunction()
{
  //In this form, it is evident that a_1 and c_1
  //are never used.
  int a_1 = 3;
  int a_2 = 4;
  int b_1 = 3;
  int c_1 = 2;
  return sum(a_2, b_1);
}
```

It is possible that a variable reference may refer to multiple versions of a variable due to conditional statements. In this case, GCC stores a list of possible versions that

may be referenced at that point in the code. There are many other complications, which are explained in part in Chapter 13 of the GCC Internals Manual (11), and in a research paper that describes the variant of SSA that the GCC developers decided to implement (12).

This is the lowest form of GIMPLE. It still contains all of the types of tuples contained in Low GIMPLE, such as function calls.

Once all GIMPLE optimization passes complete, GCC translates SSA GIMPLE into a low-level internal representation of the program called RTL.

4.4 RTL

RTL, or **R**egister **T**ransfer **L**anguage, is an LISP-like internal representation language. The program is broken up into statements called `insns`, which are very similar to assembly commands. The origin of the term *insn* is not clearly documented, and may merely be a shortening of the word "**ins**tructio**n**". The semantics of each insn is meant to be close to an assembler instruction. Hence, RTL is sometimes referred to as 'assembly-with-sugar'.

This phase of the compiler can be broken up into two parts: nonstrict RTL, and strict RTL.

4.4.1 Nonstrict RTL

SSA GIMPLE is transformed into nonstrict RTL through an expansion process. In nonstrict RTL, GCC assumes that there are an unlimited number of registers on the target machine, and optimizes the program under this assumption. This allows optimizations to modify the program without worrying about the details of register allocation.

Nonstrict RTL is eventually converted into Strict RTL.

4.4.2 Strict RTL

In strict RTL, each register referred to by an insn directly corresponds to a register on the target machine. Memory addressing changes may be made to adjust to the target platform, as well. In addition, in this version of RTL, each insn is associated with a set of machine instructions.

At this stage of compilation, we will insert the insns needed to suspend and resume each function, calculate the size of each activation record, and to align the contents of each activation record.

4.5 Assembly

Once optimizations are complete on the strict RTL representation of the program, GCC uses a description of the target machine to convert it into assembly code.

CHAPTER V

GCC Plugin Implementation

This chapter will detail our efforts to implement the parallelization process detailed in Chapter III as a GCC plugin for GCC version 4.5.2. Please note that, due to accessibility issues with the GCC codebase, this plugin was never finished. Thus, this chapter should be treated as a rough outline of how one would go about implementing our parallelization process in GCC.

The plugin must hook into three distinct phases of the compiler. First, it must hook into the parsing phase to prevent the GCC garbage collector from garbage collecting the GENERIC tree nodes that represent important runtime library `struct` definitions and functions. Second, it must register a High GIMPLE optimization pass to perform the majority of the high-level program transformations described in Chapter III. Finally, it must register a strict RTL pass to perform the low-level program transformations described in Section 3.2.9. However, before describing all of that, we will explain how the GCC plugin architecture works.

5.1 GCC Plugin Architecture

GCC provides a wide variety of places where a plugin can 'hook' into the compilation process by registering a callback function with the compiler. When the compiler reaches one of the hook points during compilation, it will call any callback functions

registered with it.

Optimization passes are a special type of hook. They must register with the GCC *pass manager*, and are required to specify, relative to another optimization pass, when it should execute.

For more information on how the plugin architecture works, see Chapter 23 of the GCC Internals documentation (11). Otherwise, the above knowledge is sufficient to gain a general understanding of the plugin described in this chapter.

5.2 GENERIC Tree Hook

The compiler plugin needs register callbacks with two hooks that occur during the parsing phase. We will explain each of these separately.

The first hook is `PLUGIN_FINISH_TYPE`. This event occurs whenever the parser finishes parsing a type into a tree node (e.g. a `struct`). Thus, the function registered with this event will execute multiple times. The goal of this hook is to save a copy of the `return_locations` struct before it is garbage collected. The callback function is given a pointer to the tree node representing the type that was just parsed. All the function needs to do is check if the name of the node is `return_locations` and, if it is, save a copy of it for use in later optimization passes.

The second hook is `PLUGIN_PRE_GENERICIZE`. This hook is executed immediately after the parser finishes parsing a file, but before the garbage collector runs to remove unreferenced nodes. The compiler plugin uses this opportunity to save a copy of the tree nodes that represent important runtime library functions that will be called by the parallelized version of the graphical dataflow program.

5.3 High GIMPLE Pass

The High GIMPLE pass is responsible for most of the program transformations described in Chapter III. It will execute after the first High GIMPLE pass, which is called `warn_unused_result`. We chose to place this pass here because the next pass begins the process of lowering High GIMPLE into Low GIMPLE.

The callback function registered for this event will be called once for each function in the graphical dataflow program being compiled, which is perfect for the program transformations described in Section 3.2. In this section, we will touch upon the changes that this callback function will make to the internal representation of each function while referring back to Chapter III.

5.3.1 Start of Function Label

First, the plugin will insert a `GIMPLE_LABEL` right before the first GIMPLE statement in the function body to denote the start of the function. Then, it will export a global symbol called `FUNCIONNAME_INIT_PC` that stores the address of the instruction that this label points to. This allows other functions that call this function, including those defined in other files, to refer back to it when initializing the program counter field for the function call's activation record. This has to be done due to the function entry point process outlined in Section 3.2.1.

5.3.2 Local Variable Representation of Activation Record Elements

Next, the plugin will define some new local variables in the function to represent important activation record items:

- An `unsigned int` that represents the `counter` and `isRunning` word.
- A `void*` that represents the saved program counter field.

- A `struct return_locations*` for each return value's `return_locations` linked list.

The strict RTL pass in Section 5.4 will eventually modify the addresses of these local variables to their proper offsets in the activation record. These variables must be marked as `volatile` to prevent the compiler from optimizing them into registers. The variables do not need to be initialized, since they should already have values in the activation record when the function begins executing.

5.3.3 Allocating and Initializing Activation Records

Now, the plugin is ready to insert the needed code to allocate and initialize activation records, following the algorithms in Section 3.2.3.1 for each scope within the function. These algorithms were carefully written so that they correspond nicely to the various GIMPLE tuples. For example, a `GIMPLE_BIND` tuple can be treated as a *scope* tree node. A description of each of these tuples can be found in chapter 12 of the GCC Internals Manual (11). Note that whenever the plugin needs to reference the size of a function's activation record, it should reference a global symbol with the name `FUNCTIONNAME_ARSIZE`. The strict RTL pass will retroactively define this symbol for every function, since this information is not available until that stage of compilation.

5.3.4 Adding Constants to Counter

The next transformation is to follow the procedure in Section 3.2.3.2 to add a constant to `counter` when needed, and to follow the procedure in Section 3.2.3.3 to launch activation records that require no arguments at the same time. These procedures should be applied to every scope within the function. Also, any modification to `counter` must be done with an atomic instruction. Luckily, GCC has a number of atomic built-in functions that are described in Section 6.49 of its documen-

tation (13). The plugin will use the `__sync_fetch_and_add` function to add a constant to the counter.

5.3.5 Writing Arguments into Activation Records

The plugin can now write arguments into the activation records that it had previously allocated by following the procedure in Section 3.2.4 for each scope in the function. Note that, once again, the plugin will need to use the `__sync_fetch_and_add` function to add a constant to the function's `counter`.

5.3.6 Waiting for Return Values

Next, the plugin will insert the code to wait for return values as described in Section 3.2.5. The atomic `compare_and_swap` function referenced in that algorithm is present in GCC as the internal function `__sync_bool_compare_and_swap`. At this compilation phase, the plugin cannot insert the required code to save the registers' states; that will need to be inserted during the strict RTL pass. It can, however, save the address of the statement that uses the unavailable return value into the program counter field of the activation record, since it previously declared a local variable that represents the address of this field. In addition, the strict RTL pass can figure out where it needs to insert the code to save the registers' states by looking for insns that modify the program counter, so no 'dummy nodes' are required to mark these locations.

5.3.7 Function Termination

Finally, the plugin should insert a call to the runtime library to terminate the function just before the `GIMPLE_RETURN` statement, as mentioned in Section 3.2.8. This allows the runtime library to deallocate the activation record, and to return the thread to the threadpool.

5.4 Strict RTL Pass

After all of the program transformations from the High GIMPLE pass are in place, the plugin is ready to perform the low level changes required to tie everything together. This pass will execute immediately after the `postreload` RTL pass, which performs register allocation and converts Nonstrict RTL into Strict RTL.

Like the High GIMPLE pass, the RTL callback function will be called once for each function defined in the graphical dataflow program that is being compiled.

5.4.1 Suspend/Resume Code

We need to create a list of registers that this function uses to determine the size and layout of the *Saved Registers' States* field of the activation record. All we need to do is perform a linear scan of the RTL statements in the function, and add any registers used in the statements to a list. We can use the order of that list as the layout of the Saved Registers' States field.

Next, at the very start of the function, we need to insert the code for the function entry point that resumes the function. Using the list and the size of each register as guidance, insert RTL statements to load the saved registers from the activation record into the actual record. The offset for each can be calculated as an offset to the stack frame pointer. At the end of all this, the pass should insert an RTL statement to jump to the address contained in the *Saved Program Counter* field of the activation record.

Finally, we can use this information to insert the appropriate RTL statements to suspend a function. Even though the RTL form of the program is close to assembly, it still links any RTL statement that refers to a variable from GIMPLE to that variable's original name. In Section 5.3.2, we defined a new local variable for the saved program counter field. Any reference to this variable is part of a suspend process. Thus, we can do a linear search for RTL statements that reference the saved program counter

field variable, and insert the RTL statements to save the function's register values into its activation record just before it.

5.4.2 Aligning Activation Records

The RTL pass will first need to align the content in each function's activation record. We can use the machine specific information available at this level to calculate the exact offset of each item in the activation record.

Thus, for each of the local variables defined in the High GIMPLE pass, as described in Section 5.3.2, we can find each reference to the variable, and change it to the proper memory location, inserting an RTL statement to load it into a register if needed.

The process goes as follows:

1. Find the first reference to the variable. It should involve a memory location, since we marked these variables as `volatile`.

2. Change the address used in the RTL statement to the correct address in the activation record. This can be calculated as a negative offset to the virtual 'stack frame' pointer that GCC uses.

3. Search the function for further references to this address, and change each to the appropriate address in the activation record in the same manner.

CHAPTER VI

Conclusion

6.1 Conclusion

Over the course of this project, we succeeded in developing a runtime model for fine-grain parallelism in graphical dataflow programs at the level of the function call. This model consists of changes to the content of each function's activation record, changing their location from the stack to the heap, and changing the logic of each function to take advantage of these changes. The runtime model offers computationally cheap synchronization among parallel function calls through a clever use of atomic instructions, such as compare-and-swap. Operating system calls are minimized by using a thread pool to manage threads that execute concurrent function invocations. Overall, this runtime model is designed to minimize the overhead required for fine-grain parallelism.

In addition, we outlined a method to implement this runtime model on a simple graphical dataflow language as an optimization pass within a compiler. This process involves an intermediate C representation that we used as a vehicle to get the graphical dataflow programs into compilers, and a runtime library that implements core features of the runtime model, such as thread pool management, activation record scheduling and activation record allocation. As a final detail, we gave an overview on how these program transformations can be performed from within GCC to implement the

runtime model in practice.

Overall, this project lays the ground work for a future project that can use the information in this report to create an implementation of the runtime model. It also opens the door to a number of other projects, which are discussed in the next section.

6.2 Future Work

6.2.1 Shifting from GCC

GCC started out as the GNU C Compiler in 1987, and eventually became the GNU Compiler Collection when it started supporting additional languages. Because it has been in active development for over 20 years, it is a large project with many components. As mentioned in Chapter IV, it has three main internal representations for programs (GENERIC trees, GIMPLE, and RTL), with different forms of each. The compiler uses a number of methods to decrease its memory footprint, including reusing GENERIC tree nodes whenever possible, which plugin developers must directly deal with. It provides a plugin architecture, which essentially consists of a collection of key header files directly from GCC's source code. Finally, its lengthy internal documentation is often short on details, and appears to be written as a quick reference guide for experienced developers. In short: writing a GCC plugin to perform program transformations on its internal representations is very difficult for people who are new to the GCC codebase. These issues prevented us from completing a functional GCC plugin to implement the proposed runtime model described in Chapter III within the time available for the project.

Future projects may wish to investigate implementing the parallelization process as a plugin for other compilers. In particular, LLVM (**L**ow **L**evel **V**irtual **M**achine) looks promising. It has extensive documentation describing how to extend LLVM, complete with examples, on its website (14). It may even be possible to write a lan-

guage frontend for LLVM for our proposed graphical dataflow language, which would avoid requiring an intermediate C representation as a vehicle for getting graphical dataflow programs into the compiler. One final advantage is that LLVM is written in C++, so it can benefit from the abstractions that that language offers.

6.2.2 Loops

This project sidesteps the issue of automatically parallelizing loops. Much work has already been done in that area using other languages (9). Future projects could look into ways to apply the large body of research on parallelizing loops to our runtime model. In particular, we know that `foreach` loops are trivially parallelizeable, and could benefit from being parallelized in a similar manner as loops in the Fortress programming language (6).

6.2.3 Runtime Library

While we describe the duties of the runtime library in this report in Section 3.3, we do not discuss its implementation in detail. We know that the runtime library needs an efficient heap allocator (such as the Linux slab allocator), a threadsafe (but fast!) queue[1] implementation for activation records that are ready to execute, and a threadpool implementation. Future projects could investigate efficient ways to implement these duties.

6.2.4 Graphical Dataflow Program Development Environment

Chapter II outlines a simple graphical dataflow language, and a C representation of programs written in it. Future projects could implement a GUI program to construct graphical dataflow programs in this language.

[1] While we say the word *queue*, the activation records have no required order of execution, so an unordered data structure would work as well.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] S. Kumar, C. Hughes, and A. Nguyen, "Architectural Support for Fine-Grained Parallelism on Multi-core Architectures," *Intel Technology Journal*, vol. 11, no. 3, pp. 217–226, Aug. 2007.

[2] D. Lea, "A java fork/join framework," 2000, [Accessed: Apr. 23, 2011]. [Online]. Available: http://gee.cs.oswego.edu/dl/papers/fj.pdf

[3] E. Archambault, J. Hogeboom, J. Montgomery, C. Pardy, C. Smith, and B. Tate, "Fine-Grain Parallelism: An Investigative Study into the merits of Graphical Programming and a Fine-grain Execution Model," *Worcester Polytechnic Institute*, Mar. 2010, Major Qualifying Project, [Accessed: Sept. 6, 2010]. [Online]. Available: https://users.wpi.edu/~lauer/MQP_protected/FinalReport_PDF-2010.pdf

[4] Free Software Foundation, Inc., "GCC, the GNU Compiler Collection," *GNU Project*, 2011, [Accessed: Apr. 26, 2011]. [Online]. Available: http://gcc.gnu.org/

[5] National Instruments Corporation, "NI LabVIEW - Improving the Productivity of Engineers and Scientists," 2011, [Accessed: Apr. 26, 2011]. [Online]. Available: http://www.ni.com/labview/

[6] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, "The Fortress Language Specification," Sun Microsystems, Inc., Tech. Rep., 2007. [Online]. Available: http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf

[7] J. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual," *Xerox Palo Alto Research Center - Systems Development Department*, Apr. 1979, [Accessed: Feb. 24, 2011]. [Online]. Available: http://research.microsoft.com/en-us/um/people/blampson/23a-MesaManual/23a-MesaManual.pdf

[8] R. Love, *Linux Kernel Development*, 2nd ed. Novell Press, 2005.

[9] M. Griebl, "Automatic parallelization of loop programs for distributed memory architectures," *University of Passau*, Jun. 2004, [Accessed: Apr. 10, 2011]. [Online]. Available: http://www.infosun.fim.uni-passau.de/cl/papers/Gri04.pdf

[10] L. Hendren, C. Donawa, M. Emami, G. Gao, J. Justiani, and B. Sridharan, "Designing the McCat Compiler Based on a

Family of Structured Intermediate Representations," *McGill University*, 1992, [Accessed: Feb. 28, 2011]. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.41.1069&rep=rep1&type=pdf

[11] Free Software Foundation, Inc., "GCC Compiler Collection (GCC) Internals," *GNU Project*, 2010, [Accessed: Sept. 6, 2010]. [Online]. Available: http://gcc.gnu.org/onlinedocs/gccint/

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, pp. 451–490, October 1991. [Online]. Available: http://doi.acm.org/10.1145/115372.115320

[13] Free Software Foundation, Inc., "GCC Compiler Collection (GCC) 4.5.2 Manual," *GNU Project*, 2010, [Accessed: Apr. 18, 2011]. [Online]. Available: http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/

[14] The LLVM Team, "Documentation for the LLVM System," 2011, [Accessed: Apr. 23, 2011]. [Online]. Available: http://llvm.org/docs/