Bug Finding Methods for Multithreaded Student Programming Projects

William Naciri

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Godmar V. Back, Chair
Ali R. Butt
Dongyoon Lee

June 23, 2017
Blacksburg, Virginia

Keywords: Binary Instrumentation, Multithreaded Programming, Debugging

Bug Finding Methods for Multithreaded Student Programming Projects

William Naciri

(ABSTRACT)

The fork-join framework project is one of the more challenging programming assignments in the computer science curriculum at Virginia Tech. Students in Computer Systems must manage a pool of threads to facilitate the shared execution of dynamically created tasks. This project is difficult because students must overcome the challenges of concurrent programming and conform to the project's specific semantic requirements.

When working on the project, many students received inconsistent test results and were left confused when debugging. The suggested debugging tool, Helgrind, is a general-purpose thread error detector. It is limited in its ability to help fix bugs because it lacks knowledge of the specific semantic requirements of the fork-join framework. Thus, there is a need for a special-purpose tool tailored for this project.

We implemented Willgrind, a debugging tool that checks the behavior of fork-join frameworks implemented by students through dynamic program analysis. Using the Valgrind framework for instrumentation, checking statements are inserted into the code to detect deadlock, ordering violations, and semantic violations at run-time. Additionally, we extended Willgrind with happens-before based checking in WillgrindPlus. This tool checks for ordering violations that do not manifest themselves in a given execution but could in others.

In a user study, we provided the tools to 85 students in the Spring 2017 semester and collected over 2,000 submissions. The results indicate that the tools are effective at identifying bugs and useful for fixing bugs. This research makes multithreaded programming easier for students and demonstrates that special-purpose debugging tools can be beneficial in computer science education.

Bug Finding Methods for Multithreaded Student Programming Projects

William Naciri

(GENERAL AUDIENCE ABSTRACT)

The fork-join framework project is one of the more challenging programming assignments in the computer science curriculum at Virginia Tech. Students in Computer Systems must manage a pool of threads to facilitate the shared execution of dynamically created tasks. This project is difficult because students must overcome the challenges of concurrent programming and conform to the project's specific semantic requirements.

When working on the project, many students received inconsistent test results and were left confused when debugging. The suggested debugging tool, Helgrind, is a general-purpose thread error detector. It is limited in its ability to help fix bugs because it lacks knowledge of the specific semantic requirements of the fork-join framework. Thus, there is a need for a special-purpose tool tailored for this project.

We implemented Willgrind, a debugging tool that checks the behavior of fork-join frameworks implemented by students through dynamic program analysis. Using the Valgrind framework for instrumentation, checking statements are inserted into the code to detect deadlock, ordering violations, and semantic violations at run-time. Additionally, we extended Willgrind with happens-before based checking in WillgrindPlus. This tool checks for ordering violations that do not manifest themselves in a given execution but could in others.

In a user study, we provided the tools to 85 students in the Spring 2017 semester and collected over 2,000 submissions. The results indicate that the tools are effective at identifying bugs and useful for fixing bugs. This research makes multithreaded programming easier for students and demonstrates that special-purpose debugging tools can be beneficial in computer science education.

Dedication

To the loving memory of my grandfather Lowell Kenyon Good

I will always remember our fantastic adventures

Acknowledgments

I would first like to thank my advisor Dr. Godmar Back. His hands-on teaching style and excellent communication skills have been a catalyst for my development. I am truly grateful to have been his student.

I would also like to thank Dr. Ali Butt and Dr. Dongyoon Lee for serving on my committee and providing insightful suggestions for improvement.

I would also like to thank the wonderful Computer Science Professors at Virginia Tech for everything I have learned in the classroom. In particular, Dr. William McQuain always captured my attention and kept me on the edge of my seat in his lectures. Additionally, I want to thank the department for supporting me with a GTA position.

Last but not least, I am grateful for the love and support from my parents. I could not have done it without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern day processors feature an increasing number of cores each year. As a result, parallel programming has become ubiquitous. A key paradigm in parallel programming is multithreading. A thread is a unit of sequential execution that can be scheduled to a distinct core. A multithreaded application can experience a performance speedup when multiple threads execute concurrently. Although threads have remarkable performance advantages, they are difficult to program. For that reason, educators have been determined to improve their curriculum related to threads [22]. In this research, we attempt to make learning multithreaded programming easier with a special-purpose debugging tool.

1.1 Multithreaded Student Project

At Virginia Tech, undergraduate computer science students are required to take an upper-level computer systems class. This is usually the first time they are exposed to multithreading. In the second programming project, students must create a framework to support fork-join tasks. The goal of a fork-join framework is to provide an interface for divide-and-conquer algorithms to execute recursive tasks in parallel.

Implementing the fork-join framework teaches students about thread management and synchronization primitives. However, many students are quickly overwhelmed by this project because multithreading requires programmers to think in a way humans find difficult [55]. For that reason, multithreaded programming is usually taught as a specialized topic at senior and graduate levels in research universities. Even experts feel that threads are very hard to program [35, 27]. The main challenge of programming threads is synchronizing their shared memory accesses. Since each thread has its own control flow, the programmer must ensure that shared variables are accessed in the correct order.

1.2 Motivation

When working on the fork-join framework project, students face obstacles such as deadlock, lifelock, data-races, and ordering violations [37]. Additionally, students must conform to the specific semantic requirements of the project to ensure their implementation produces correct results. With such a variety of possible bugs, the majority of time spent on this project is for debugging.

Existing debugging tools focus strictly on data-races. For instance, Helgrind is a thread error detector that reports when improper synchronization is observed in an execution. Although it detects data-races well, it does not identify and describe the high-level root cause of bugs. In order to give students a better understanding of their bugs and how to fix them, we propose a custom debugging tool designed specifically for this project.

Willgrind is a special-purpose debugging tool for the fork-join framework project that performs dynamic program analysis. Its goal is to improve upon pure data-race detectors by leveraging program-specific information. With knowledge of the project semantics, the tool can directly detect bugs and provide high-level descriptions about their cause. On the other hand, data-race detectors can only inform students about conflicting memory accesses.

1.3 Core Contributions

This research is driven by feedback from past students who expressed difficulty working on the fork-join framework project. We improved upon existing classroom tools and methods for debugging this challenging programming assignment. The contributions of our approach are:

Willgrind To make the fork-join framework project easier, we created a custom tailored debugging tool called Willgrind. It is built from scratch using the Valgrind framework and instruments student binaries to check for bugs at run-time. In order to detect bugs, the characteristics of the correct execution are defined as a model. Bugs are reported when the student program violates this model.

WillgrindPlus To increase bug detection, we extended the tool to check for potential failures. Since bugs may be hidden in certain thread interleavings, we created WillgrindPlus, a second tool that leverages logical clocks to observe happens-before relationships. By detecting missing happens-before relationships between events in a student program, WillgrindPlus can identify regions of code that lack synchronization. Missing synchronization indicates that the student program is vulnerable to a bug that can manifest itself in a different execution.

Deadlock Detection Deadlock arises in the fork-join framework when multiple threads have a lack of progress because they are blocked indefinitely in a system call. This is difficult to detect because it is not known if/when these threads may be unblocked. Our deadlock detection heuristic accurately identifies when a thread is blocked indefinitely by leveraging program-specific knowledge. We use the insight that there are no system calls in the fork-join framework that block for extended periods of time. Therefore, it is safe to assume that when threads are blocked in a system call for more than a few seconds they will never unblock because deadlock is present.

Web Interface Helgrind output is sometimes difficult to read and interpret. Willgrind and WillgrindPlus provide output through an interactive web interface that improves on Helgrind by providing meaningful messages, more stack traces, and a high-level description of bugs. After running the selected tool, results are posted to a cloud-based portal where students can view the analysis of their program confidentially. Not only is this system helpful for students, but it is also useful for researchers to analyze student submissions.

User Study In order to evaluate the effectiveness and usefulness of the tools, we performed a user study on Virginia Tech students who worked on the fork-join framework project during the Spring 2017 semester. The tools were provided to the students and the results show that a plethora of different bugs were detected in their submissions. Additionally, we performed a voluntary survey on the students' experiences with the tools that further validate our results.

1.4 Roadmap

Chapter 2 provides background on the fork-join framework project and the Valgrind framework. Chapter 3 describes the design and technical details of Willgrind. Chapter 4 provides background on happens-before based checking and describes the implementation of WillgrindPlus. Chapter 5 evaluates the tools through a user study. Chapter 6 compares related work. Finally, Chapter 7 describes future work and our conclusion.

Chapter 2

Background

This chapter provides background information to understand the concepts underlying the design of Willgrind. We assume the reader has basic knowledge of multithreaded programming.

2.1 Fork-Join Parallelism

In divide and conquer algorithms, problems are broken up into smaller subproblems that are solved recursively to generate a solution [50]. The subproblems are usually independent and can be solved separately, which allows them to be parallelized using a thread framework. The framework is responsible for mapping distinct subproblems or tasks to threads. While several types of frameworks exist, the fork-join framework provides efficiency, simplicity, and regularity [34]. In fork-join parallelism, a program divides (forks) into two or more subprograms that can be executed in parallel and combined (join) back to a single program [23]. Furthermore, strict fork-join parallelism is a simplification that requires each fork to be classified as a child of another fork. These forks form an execution hierarchy with beneficial properties such as composability.

Figure 2.1 illustrates the fork-join flow of a toy Fibonacci algorithm. The root task computes the fourth Fibonacci number by forking into several subtasks which are joined to generate the result.

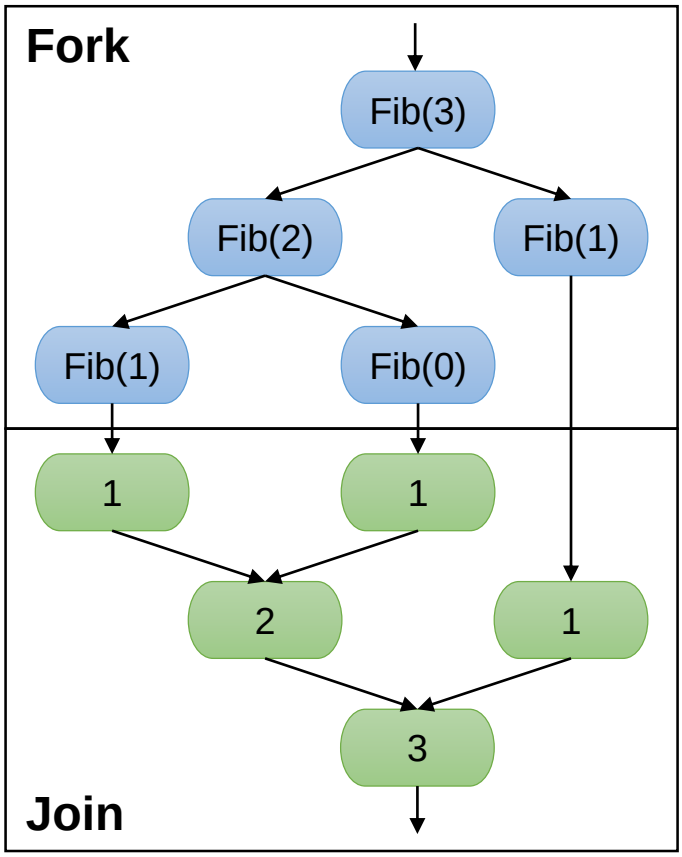

Figure 2.1: A fork-join flow

2.1.1 Fork-Join Framework

Spawning a new thread for each forked task creates prohibitively high overhead because creating and destroying threads is relatively expensive. Lea describes the Java fork-join framework [34] that avoids unnecessary overhead by managing a fixed pool of worker threads.

In the Java fork-join framework, forked tasks are represented by a Callable or Runnable subclass. These objects are termed fork-join tasks and represent a unit of work. The primary

duty of the framework is to manage the distribution of tasks to a pool of worker threads. Conversely, worker threads are absolved of any high-level synchronization demands of the program.

Worker threads follow a simple routine of retrieving tasks from a queue and executing them. Tasks are created for workers on a global queue. Additionally, each worker has its own queue that is used to store any subtasks that it spawns. A worker should finish subtasks on its local queue before retrieving more work from the global queue. Tasks are popped off the queues in LIFO order which leads to efficient execution when there is plenty of work. However, when a worker does not have any local tasks and the global queue is empty, the worker is said to be idle and a heuristic is needed to keep it busy.

For the framework to achieve high performance, threads should not idle while others are working. Idling is minimized with a strategy first introduced by the Cilk work-stealing scheduler [18]. The work-stealing strategy empowers idle workers to stay busy by stealing tasks from a coworker's queue. Stealers operate on the opposite side of the queue as the queue's owner to avoid contention. Additionally, stealing in a FIFO order provides the stealer with older and thus larger tasks to stay busy.

2.1.2 Fork-Join Framework Programming Project

Introduction to Computer Systems is a cornerstone course of the Computer Science curriculum at Virginia Tech. Students learn about a variety of operating system topics including multithreaded programming. In the classroom, they learn about the Java fork-join framework and outside the classroom, they recreate it in C.

This programming project teaches students about thread management and synchronization primitives. The end result is a fork-join framework that facilitates the shared execution of dynamically created tasks. In addition to correctness, students are graded on their framework's ability to enhance the performance of divide-and-conquer algorithms called client

programs.

The starting materials provide API declarations in a header file and students must complete the definitions. Because these definitions are usually only a few hundred lines of code, the project's difficulty is frequently underestimated. There are numerous types of bugs that can occur from incorrect thread safety or violating the specific semantic requirements of the project. With that being the case, a majority of time spent on this project is for debugging.

Future

According to the Java language specification, "A Future represents the result of an asynchronous computation [20]." Similarly, in this project a future is a placeholder or abstraction for a to-be-completed fork-join task. When a client submits work to the fork-join framework, a future is returned as a receipt. The future must be redeemed with the framework to obtain the result of the work.

The project header file declares `struct future` for use by the API but does not define it. Students have the freedom to implement it however they choose. Nonetheless, it is intended to encapsulate fork-join tasks with their result. Additionally, the header file defines a fork-join task as a pointer to a function in the client program that receives data to be computed. Listing 1 shows the future declaration and the function pointer type.

```
struct threadpool;
struct future;
typedef void * (* fork_join_task_t) (struct thread_pool *pool, void * data);
```

Listing 1: Definition for fork-join task

Submit

Students implement the fork-join framework as an interface for a sequential client program to execute tasks in parallel. To submit work, the client provides a task and data to the

framework. In return, the client receives a future from the framework. On the other end, the framework allocates a future and places it on the global queue. The future is executed asynchronously from the client and the result of its computation is returned when the client redeems the future. Additionally, worker threads may internally submit work in the form of subtasks. In this case, a future is placed on the worker's local queue. Listing 2 shows the declaration of `threadpool_submit`.

```
/*
* Submit a fork join task to the thread pool and return a
* future. The returned future can be used in future_get()
* to obtain the result.
* 'pool' - the pool to which to submit
* 'task' - the task to be submitted.
* 'data' - data to be passed to the task's function
*
* Returns a future representing this computation.
*/
struct future * thread_pool_submit(struct thread_pool *pool, fork_join_task_t task, void * data);
```

Listing 2: Declaration for threadpool_submit

Get

In order to obtain the result of a future, the client calls `future_get`. At this point, the framework returns the result if it has been computed. Otherwise, work-helping can be performed. This is a strategy where the obtaining thread completes the future's task on behalf of another worker. However, work-helping is not always required. For instance, if the task has already been started, the obtaining thread should block until another worker completes the task. However, under no circumstances should `future_get` return before the result has been computed. Listing 3 shows the declaration of `future_get`.

Free

Once the client has obtained the future's result, the future is no longer necessary. The client must call `future_free`, shown in Listing 4, to indicate that it is done using the future. At

this point, the framework will reclaim the future's memory.

```
/* Make sure that the thread pool has completed the execution
* of the fork join task this future represents.
*
* Returns the value returned by this task.
*/
void * future_get(struct future *);
```

Listing 3: Declaration for future_get

```
/* Deallocate this future. Must be called after future_get() */
void future_free(struct future *);
```

Listing 4: Declaration for future_free

2.2 Valgrind Framework

Both static and dynamic program analysis are used by developers to catch bugs. For instance, memory leaks can be detected statically [26] or dynamically [41]. In static analysis, the program's source code is inspected and in dynamic analysis the program's execution is inspected. The key difference is that the results of static analysis hold true for all possible executions, whereas the results of dynamic analysis only hold true for certain executions [29]. However, static analysis is limited because it cannot identify run-time bugs. Therefore, static analysis is not practical for run-time features such as dynamic binding, polymorphism, and threads [19]. The latter is the primary aspect of this research and for that reason we have chosen to use dynamic program analysis for bug detection.

A variety of techniques can be used to perform dynamic program analysis, all of which require a means for observing the program's execution. A common technique is instrumentation which inserts foreign code into an application to observe its execution. Instrumentation can be performed at compile-time [33], linking-time [40], or run-time [38].

Our tool performs dynamic program analysis using Valgrind [42], a dynamic binary instrumentation (DBI) framework. It provides an environment for tools to instrument and add

analysis code to an original client program. Furthermore, Valgrind does not require source code because it operates directly on client binaries.

2.2.1 Architecture

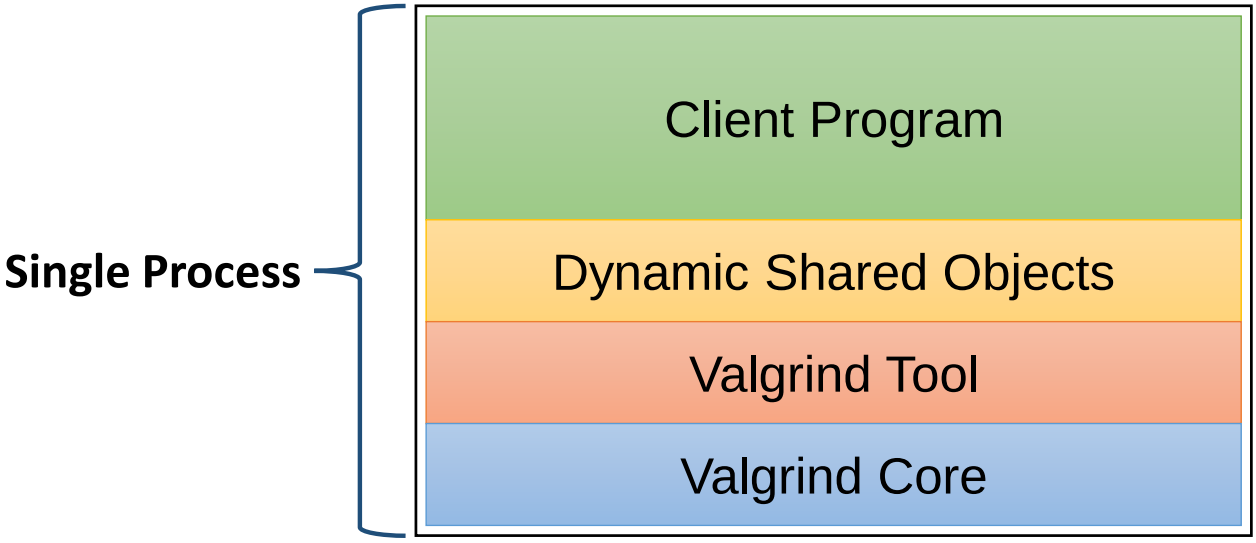

Figure 2.2: Valgrind framework layers

Valgrind uses dynamic binary recompilation to instrument client programs. This process involves disassembling the client binary into an intermediate representation (IR), instrumenting the IR, and reassembling the IR into machine code. The instrumentation step provides an opportunity for Valgrind tools to add and remove IR statements for the purpose of obtaining analysis information at run-time.

Valgrind Supervision

In addition to instrumentation, binary recompilation is necessary to gain control over the client program. During recompilation, Valgrind alters the binary so that every client instruction is run under the supervision of its framework. This gives tools the ability to record any event in the client program.

In previous versions, Valgrind gained control of the client by using the dynamic linker to inject its framework as a shared object into the client program. This approach was unreliable because it is possible for some client code to run natively before the framework. Instead, Valgrind runs first, as its own program, and then loads the client as a guest within the same process. Figure 2.2 shows the layers of the Valgrind framework.

Since the client program is squeezed into the same process as Valgrind, they must share many of the same resources such as registers and memory. At startup, several subsystems such as the address space manager, scheduler, and translator are initialized to facilitate running the client program as a guest.

2.2.2 Instrumentation & Execution

When under the supervision of Valgrind, a client program is never executed directly. Instead, it is treated like source code in the recompilation process. Essentially, this process is performed in a just-in-time fashion with three different steps: translation, instrumentation, and execution.

Phase 1: Translation

The first step of the Valgrind pipeline is translating client machine code into IR. The IR is composed of a sequence of RISC-like statements that resemble x86 instructions. The reason behind using IR is that it is more convenient to translate and instrument than the x86 instruction set. Listing 5 shows the translation of an indirect function call to a sequence of IR statements.

Translations are performed at the granularity of basic blocks, which are single-entry multiple-exit stretches of code. At the end of each basic block, a jump statement is added to transfer control back to Valgrind. At this point, Valgrind's dispatcher fetches the next basic block from the client binary or from the translation cache if it has been run before.

```
0x40054B: callq *%rax

------ IMark(0x40054B, 2, 0) ------
t7 = GET:I64(48)            # get %rsp
IR-NoOp                     # noop
PUT(48) = t7                # put %rsp
t14 = Sub64(t7,0x80:I64)    # get function pointer
PUT(184) = t14              # put %rip
```

Listing 5: Indirect function call translated to IR

Phase 2: Instrumentation

After converting machine code into IR, Valgrind gives the tool an opportunity to instrument each basic block before execution. The easiest way to instrument the IR is to insert calls to C functions before or after instructions of interest. Alternatively, IR can be inlined with the basic block to reduce overhead at run-time. Listing 6 shows the instrumentation of an indirect function call. Just before the indirect function call, a call to an analysis function is inserted. The analysis function is located in the tool and receives the indirect function's address as an argument. This acts as a hook into indirect function calls in the client program. Indirect function calls are significant in the fork-join framework because fork-join tasks are called using function pointers.

```
------ IMark(0x40054B, 2, 0) ------
t7 = GET:I64(48)                                        # get %rsp
IR-NoOp                                                 # noop
PUT(48) = t7                                            # put %rsp
t14 = Sub64(t7,0x80:I64)                                # get function pointer
DIRTY 1:I1::: handle_indir_fn[rp=1]{0x58007cd0}(t14)    # instrumented
PUT(184) = t14                                          # put %rip
```

Listing 6: Indirect function call instrumented with analysis function

Phase 3: Execution

After the tool has finished instrumenting, Valgrind optimizes the IR and reassembles it back to machine code. Besides the instrumented code, the final machine code guarantees the same

behavior as the original client program. However, simply executing this code could cause Valgrind to lose control of the process.

The problem lies in the fact that Valgrind and the client share the same process, thus there is only one set of registers. Since Valgrind is the supervisor, it has the responsibility to manage the client's control flow separate from its own. This is accomplished by conceptually running the client program on a guest CPU, meaning a second set of registers is maintained for the client. Just before the client runs, Valgrind saves its own registers and emulates the client registers. Additionally, prior to control returning to Valgrind, the client registers are saved and Valgrind's registers are restored.

The synthetic guest CPU is quite transparent to the client. Furthermore, the value of special-purpose client registers, such as the stack pointer, are preserved for tools to inspect the client state. Although, some of the client's general-purpose registers may spill over to memory. A drawback of this approach is that subtle bugs may occur from running standard libraries on both the guest and host CPU. The primary reason for this is that standard libraries and Valgrind are both designed to interact with the kernel directly. Consequently, tool developers cannot use standard libraries. In our tool, we utilize data structures from the Pintos project [45] because they are void of any standard libraries.

2.2.3 Function Redirection

As an alternative to direct instruction instrumentation, Valgrind provides function redirection as a means to instrument entire functions. This mechanism can intercept client functions to manipulate control flow or obtain analysis information. If the function's name is known, Valgrind can override it by leveraging the shared memory space with the client and using the dynamic linker to inject a function replacement.

Function redirection is particularly useful for instrumenting a known API like the fork-join framework. Once the specified function is intercepted, the replacement can act as a wrapper

or even alter the function's behavior. Typically, tools supply function replacements that examine arguments, call the original function, and inspect the result. Since the function replacement runs in the client program, Valgrind provides a client request feature. This is a trapdoor mechanism to send information from the function's replacement to the tool.

Listing 7 shows a wrapper for `future_get`. Client requests are inserted before and after the original function call to redirect control flow to analysis functions in the tool. The analysis functions are called handlers and are used to examine the argument before the original function call and inspect the return value after the original function call. Listing 8 shows the effects of the wrapper in C.

```
void* I_WRAP_SONAME_FNNAME_ZU(NONE, future_get)(void* future) {
  void* result;
  OrigFn fn;
  VALGRIND_GET_ORIG_FN(fn);
  VALGRIND_DO_CLIENT_REQUEST_STMT(CLREQ_FUTURE_GET_PRE, future, 0, 0, 0, 0);
  CALL_FN_W_W(result, fn, future);
  VALGRIND_DO_CLIENT_REQUEST_STMT(CLREQ_FUTURE_GET_POST, future, result, 0, 0, 0);
  return result;
}
```

Listing 7: Function wrapper for future_get

```
void* future_get_replacement(void* future) {
    void* result;
    handle_future_get_pre(future);
    result = future_get(future);
    handle_future_get_post(future, result);
    return result;
}
```

Listing 8: Effects of function wrapper for future_get

2.2.4 Multithreading in Valgrind

Threads are an important consideration when designing a DBI framework. To fully support multithreaded applications, a framework and its tools must be thread safe. Ensuring thread safety can become a burden, especially for memory intensive instrumentation.

In Valgrind, multithreaded applications run properly, just not in parallel. Valgrind and its tools run as a single thread and the execution of client threads is serialized. Essentially, each thread is correctly abstracted and registered with the kernel but only one runs at a time.

The serialization is performed using a big lock model. A thread must acquire the big lock to run and drop the big lock before it enters a blocking system call, or after it has reached a quantum (100,000 basic blocks). The big lock is implemented as pipe-based lock by default. Pipe-based locks are generally unfair and can lead to starvation. Alternatively, a futex-based ticket lock is available on certain platforms if the `--fair-sched` flag is specified by the user.

Other DBI frameworks [38] do not serialize threads and Valgrind plans to remove the big lock in future work. While it provides simplicity and convenience for tool developers, the big lock contributes to performance shortcomings in Valgrind. As a result, debugging the fork-join framework project with a Valgrind tool can be extraordinarily slow.

Idle CPU State

Energy efficiency is a growing concern for operating systems. The kernel must provide equal processor time for all threads while also minimizing power consumption. To achieve this goal, the kernel distributes the execution of threads onto different cores. Furthermore, when a thread is not running, the kernel decreases the frequency of its core to save power. This low power state is called the C-state [49] and is commonly referred to as the idle CPU state.

Although the idle CPU state is beneficial for saving energy, there is a latency for a core to become operational again upon exiting the state [46]. This latency interacts poorly with Valgrind's thread serialization. Since only one thread runs at a time, the other threads are always blocked. Therefore, the cores of the blocked threads are likely idle and in the low power state. When a thread eventually unblocks, its core can experience a warmup period to exit the low power state resulting in degraded performance. We evaluate the interaction

of thread serialization and the idle CPU state in Section 5.1.1.

Chapter 3

Willgrind

Valgrind provides the low-level infrastructure to support program instrumentation. The role of a Valgrind tool is to utilize the framework suitably in order to analyze a target program. This chapter first describes how the Willgrind tool achieves its program analysis. Then we discuss how the tool is implemented and its interaction with Valgrind. Next, we compare solutions for deadlock detection. Finally, we describe the web interface that Willgrind uses to output results.

3.1 Design

Shadowing is a common technique used by dynamic program analysis tools [41, 44, 6] to detect bugs at run-time. During execution, program values of interest are tracked with an associated shadow value in the tool. The shadow values contain useful information to identify bugs. Willgrind uses shadowing to model the correct execution of a fork-join framework and verify that the target program obeys this model. In this section, we describe the approach used to detect bugs.

3.1.1 Invariants

In order to detect bugs, a tool must clearly define their characteristics. We accomplish this by defining the reverse: the characteristics of an execution that is bug-free. These are known as invariants, or conditions that must hold true during the correct execution of a program. Identifying invariant violations can be used as a strategy to verify whether program is correct. However, it is difficult to define a comprehensive list of invariants because many programs obey invariants that may not even be known to the authors of the code [24]. This is true in our case, as many characteristics of a correct fork-join framework are not documented in the project specification [4] or tested in the grading script. The fork-join framework API only describes a few abstract, high-level properties because students are given freedom in how they choose to implement low-level details. Although the grading script is rigorous and covers many test cases, it is still limited by its lack of sight into the fork-join framework. A DBI tool is necessary because it has the ability to inspect the program state at any time during execution.

Invariants are not always obvious. Initially, we missed some but through this research we created a thorough list of invariants that exist in a bug-free fork-join framework. To define our invariants, we used the insight that every function in the fork-join framework API has one thing in common: they all involve a pointer to `struct future`. Thus, we can follow the future throughout each API call and model its state. An incorrect state would be the result of a invariant violation.

3.1.2 Model Futures

The Memcheck tool uses shadow memory [41] to track a memory location. Shadow memory is a value pertaining to a memory location, maintained by the tool, to describe its state. Similarly, in Willgrind we need to know the state of futures. Since each fork-join framework may implement `struct future` differently, we cannot directly interpret the state. Instead,

futures are shadowed with our own **model future** construct. A model future contains a state, history of states, and debugging information. For each future in a fork-join framework, a model future is created in Willgrind.

The states of model futures shown in Table 3.2 are based off of the fork-join framework events in Table 3.1. Associating a before and after event with a state is important because this is when bug checking occurs. *Being Worked On* does not have a visible after event and *Complete* does not have a visible before event. Even though *Free* has a visible after event, it is not used for reasons explained in Section 3.1.5.

Name	**Event**
Submit Pre	Immediately before `threadpool_submit` is called
Submit Post	Immediately after `threadpool_submit` is returns
Task Start	Immediately before the future's task is executed
Task Finish	Immediately after the future's task is executed
Get Pre	Immediately before `future_get` is called
Get Post	Immediately after `future_get` returns
Free Pre	Immediately before `future_free` is called

Table 3.1: Fork-join framework events

State	**Before Event**	**After Event**
Born	Submit Pre	Submit Post
Being Worked On	Task Start	–
Complete	–	Task Finish
Obtained	Get Pre	Get Post
Free	Free Pre	

Table 3.2: Model future states

3.1.3 Model Future Life Cycles

Model futures follow a rigid order of states in their lifetime: *Born*, *Being Worked On*, *Complete*, *Obtained*, then *Free*. It is possible for before and after events to be interleaved across multiple states. As a result, the transitions between states can vary in different executions. This is an effect of work-stealing and work-helping. Figure 3.1 illustrates the different transitions in a state machine. There are exactly six possible transition sequences:

1. Submit Pre → Submit Post → Get Pre → Task Start → Task Finish → Get Post → Free Pre
2. Submit Pre → Submit Post → Task Start → Get Pre → Task Finish → Get Post → Free Pre
3. Submit Pre → Submit Post → Task Start → Task Finish → Get Pre → Get Post → Free Pre
4. Submit Pre → Task Start → Submit Post → Get Pre → Task Finish → Get Post → Free Pre
5. Submit Pre → Task Start → Submit Post → Task Finish → Get Pre → Get Post → Free Pre
6. Submit Pre → Task Start → Task Finish → Submit Post → Get Pre → Get Post → Free Pre

The variation of state transitions is the result of multiple threads working on the same future. For instance, in #6, *Task Finish* occurs before *Submit Post*. Even though `threadpool_submit` has not returned, its effects are visible when it places the future on a queue. At this point, other threads may steal the future and complete it before the submitter returns.

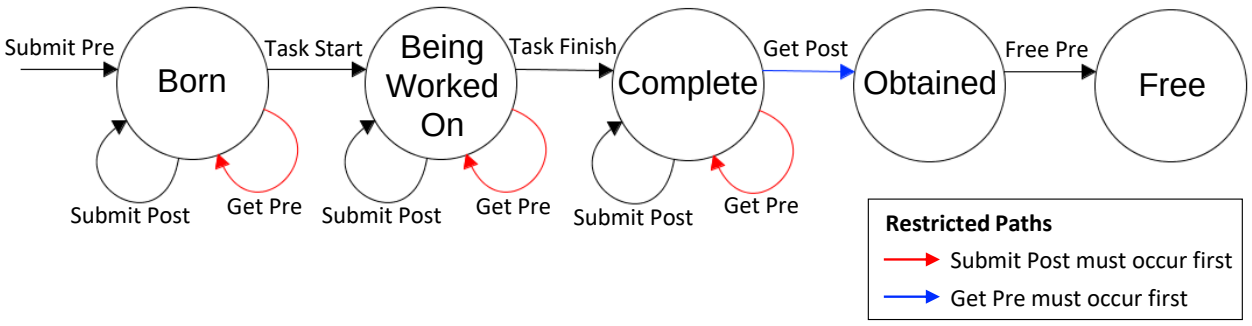

Figure 3.1: State transitions for a model future

3.1.4 Born State

Willgrind stores model futures in a table and indexes them by the address of the future. However, using the future's address as the sole lookup key is insufficient because the address is not always available. Specifically, the future's address is not known until `threadpool_submit` returns.

As shown in the definition of `threadpool_submit` in Listing 2, the caller must provide a task and data for the function to return a future. The fact that the future does not exist at the time `threadpool_submit` is invoked, can create an issue for certain executions. Specifically, if `threadpool_submit` has made the future visible to other threads but not yet returned, another thread can start the future. In this case, Willgrind would not be able to record the *Task Start* event because a model future has not been created. Therefore, the model future must be created as soon as it is known that a new future is being created, even if `threadpool_submit` has not yet returned.

In order to access these model futures whose future address is not yet known, there needs to be a way to retrieve them. This is accomplished with a second table that indexes model futures by their task and data pointers. The second table circumvents the lookup problem by allowing model futures to be retrieved by their task-data pairs, which are known when `threadpool_submit` is called. Once `threadpool_submit` returns, the task-data pair is no longer needed for lookup and the model future is indexed by the future's address.

This approach is made possible with the assumption that the same task and data pointers will not be submitted multiple times, at least not until the first task completes. A task pointer may be reused if it calls itself recursively but in this case, the data pointer will always be different. As a corollary, we can assume there will be no task-data pair duplicates as long as they are removed from the table once `threadpool_submit` returns.

3.1.5 Memory Reuse

Any DBI tool that tracks memory addresses needs to consider memory reuse. When memory is deallocated, it can be recycled quickly and the same address may be reused for another allocation. It is critical that tools understand the lifetime of the memory they track.

A common instance of memory reuse in the fork-join framework is when a future is deallocated and its address is quickly reallocated for another future. This situation is especially frequent in implementations that use a freelist to allocate futures. In order to prevent any false positives caused by such memory reuse, all model futures must be destroyed before their future is deallocated. As a result, Willgrind cannot record a *Free Post* event because it occurs after future deallocations. Furthermore, Willgrind does not does not detect use-after-free bugs; however, there are many other tools, such as Memcheck, that focus on these types of bugs.

Initially, when Willgrind did not consider memory reuse, model futures were destroyed after `future_free` returned. Consequently, we observed conspicuous false positives, such as this illegal transition sequence:

- Submit Pre → Submit Post → Task Start → Task Finish → Get Pre → Get Post → Free Pre → **Submit Pre** → Free Post

Obviously, a future cannot be submitted again after it has already been freed. This false positive occurred due to the fork-join framework implementation reusing the same memory

(and thus the same address) for a future before the *Free Post* event was observed.

3.1.6 Model Future Checking

Since all fork-join framework events involve a pointer to a future, they are convenient points to update model futures and check for bugs. The process of updating model futures is called model maintenance and the first step is using the future's address to retrieve the corresponding model future. Next, a state transition is performed on the model future based on the event. This involves saving the old state, capturing debugging information, and updating to the new state. After model maintenance, the model future is synchronized with the real future and is ready for the second step called bug checking. It is important to perform model maintenance before bug checking so any bugs can be reported with the latest debugging information.

Event	**Bug**	**Name / Type**
Submit Pre	None	None
Submit Post	State is not *Born*, *Being Worked On*, or *Complete*	Bug 3 / Order violation
Task Start	Task has already started	Bug 5 / Semantic violation
	State is not *Born*	Bug 1 / Order violation
Task Finish	State is not *Being Worked On*	Bug 2 / Order violation
Get Pre	State is not *Born*, *Being Worked On*, or *Complete*	Bug 6 / Order violation
Get Post	Task has not been called	Bug 4 / Semantic violation
	State is not *Complete*	Bug 7 / Order violation
Free Pre	State is not *Obtained*	Bug 8 / Order violation

Table 3.3: Detected future bugs

During bug checking, the model future state machine is used to detect illegal transitions. If the observed fork-join framework event does not correspond to a legal state transition, a bug is reported. Additionally, some bugs are separated into a semantic violation category to provide a more specific cause of the bug. Table 3.3 shows the bugs that are checked at each

event. Deadlock is detected differently and is discussed in Section 3.3.

3.2 Implementation

Willgrind is composed of run-time code, instrumentation code, and function replacements. While the run-time and instrumentation code both run on the host CPU, the function replacements run in the guest space. This section describes the tool's means of instrumentation and detecting bugs at run-time.

3.2.1 Function Wrapping

Valgrind function redirection provides an interception interface for functions whose names are known in advance. Since the names of functions in the fork-join framework API are known we use function redirection to instrument them. However, the names of fork-join task functions are not known in advance, so they are instrumented differently as discussed in Section 3.2.2.

We define function wrappers for each function in the fork-join framework API. For the wrapper to become active, it must be present in the same text section as the function it wraps [43]. In other words, the function wrapper must be located in the client space. During start up, Valgrind reads the symbol table for functions with the wrapper prefix and notifies the translator to redirect to the wrapper during execution. Additionally, Valgrind provides a way for tools to specify files they want to preload into the client in order to inject the wrappers in the client space.

Since, wrappers must run in the client program, we use Valgrind client requests to send analysis information back to the tool. The client requests are received by the Valgrind core and then sent to the tool's run-time code for handling.

Each wrapper captures the arguments of the function and passes them in a *pre* client request.

Next, the original function is called. Finally, the return value is captured and sent in a *post* client request.

On the other end, when Willgrind receives a client request model maintenance and bug checking is triggered. During model maintenance the model future is updated and debugging information is saved. To obtain debugging information, the active guest thread ID is retrieved from Valgrind. Next the stack trace is captured as an array of addresses. Finally, the model future's state is recorded in the shadow thread's history which is explained in Section 3.2.5.

3.2.2 Basic Block Instrumentation

The primary part of the instrumentation code is the `wg_instrument` function. This is a callback function registered with the Valgrind core that receives a basic block of IR, instruments it, and returns it back to the core. The instrumentation involves inserting calls to analysis functions and also reading guest registers. Even though the analysis functions execute at run-time, they run on the host CPU and cannot directly access the guest state. The guest must be compelled to give its information to the analysis functions through instrumentation.

While function redirection is convenient, it cannot be used to instrument fork-join tasks. This is because they are pointers to functions whose names are not known in advance. For that reason, direct binary instrumentation is required. The model futures need to be maintained before and after each task is executed. To accomplish this Willgrind directly instruments machine code in the form of IR.

In this task we use a heuristic to detect fork-join task calls and returns and also insert analysis functions called handlers. We start by identifying that task function calls are always indirect.

3.2.3 Instrumenting Fork-Join Task Calls

Detecting indirect function calls is straightforward. They are always found at the end of a basic block. Since the target address of the call is computed at run-time, the last instruction of the basic block is an assignment of a temporary to the guest program counter, whereas a normal function call is an assignment of an address to the program counter.

After detecting indirect function calls, they must be instrumented by inserting a handler before them. The IR provides a convenient instruction, `DIRTY`, that can redirect the control flow by calling a handler in the tool. This is minimally intrusive to the client because only the function call is added. The handler itself runs on the host CPU.

`DIRTY` allows the instrumenter to specify up to three arguments for the function call. In our case, we will use all three to pass the indirect function address, the data, and the current guest stack pointer:

Indirect function address and data: We have established that all fork-join tasks are indirect function calls, but not all indirect function calls come from fork-join tasks. For this reason, we must verify that the function address actually belongs to a task. In fact, we already keep a record of tasks submitted in the task-data pair table from Section 3.1.4. This allows us to conveniently identify an indirect function call as a fork-join task and also associate it with the corresponding model future.

Stack Pointer: We retrieve the stack pointer because it will be used to identify when the task returns. During instrumentation, guest register `RSP` is assigned to an IR temporary. At run-time, the temporary is read and passed to the handler. As a result, Willgrind gains knowledge of tasks' stack pointers. Since each thread has its own stack, the handler defers to the thread module (Section 3.2.5) to store stack pointers. The currently running thread ID is queried and passed to the thread module with the stack pointer.

Once an indirect function call is identified as a fork-join task, its corresponding model future is retrieved to perform model maintenance and bug checking. Willgrind checks the model future state and also makes sure the task has not been called previously.

3.2.4 Instrumenting Fork-Join Task Returns

To detect fork-join task calls, we only needed to instrument indirect function calls. However, to detect fork-join task returns, we must instrument all function returns. Additionally, for calls, we simply instrumented at the very end of the basic block. This will not work for function returns. Even though returns are always located at the end of a basic block, instrumentation must take place before the return is executed. Instrumenting after the return instruction will be too late because the stack pointer will already be updated to the saved stack pointer of the previous frame. The guest information of interest is the stack pointer for the current basic block which will be used to identify fork-join task returns.

Recall that when a fork-join task is called, Willgrind saves the stack pointer. Our heuristic checks the guest register `RSP` before all return instructions to identify if a fork-join task is returning. Figure 3.2 shows an example of a fork-join task with recursive calls on a thread's stack. A given task has returned when the current stack pointer is that of the saved stack pointer.

First the return instruction must be identified. The IR contains `IMark` instructions that serve as hints to the instrumenter. They represent the location of the original x86_64 instructions in the IR and also provide the original opcode. Willgrind checks each `IMark` instruction to identify the location of original `retq` instructions. This is a simple opcode check for `0xC3`. Once an original return instruction is found a `DIRTY` instruction is added, just before, to call Willgrind's function return handler. Additionally, guest register `RSP` is assigned to an IR temporary and passed as an argument to the handler.

At run-time, the instrumentation produces a call to the return handler whenever a function

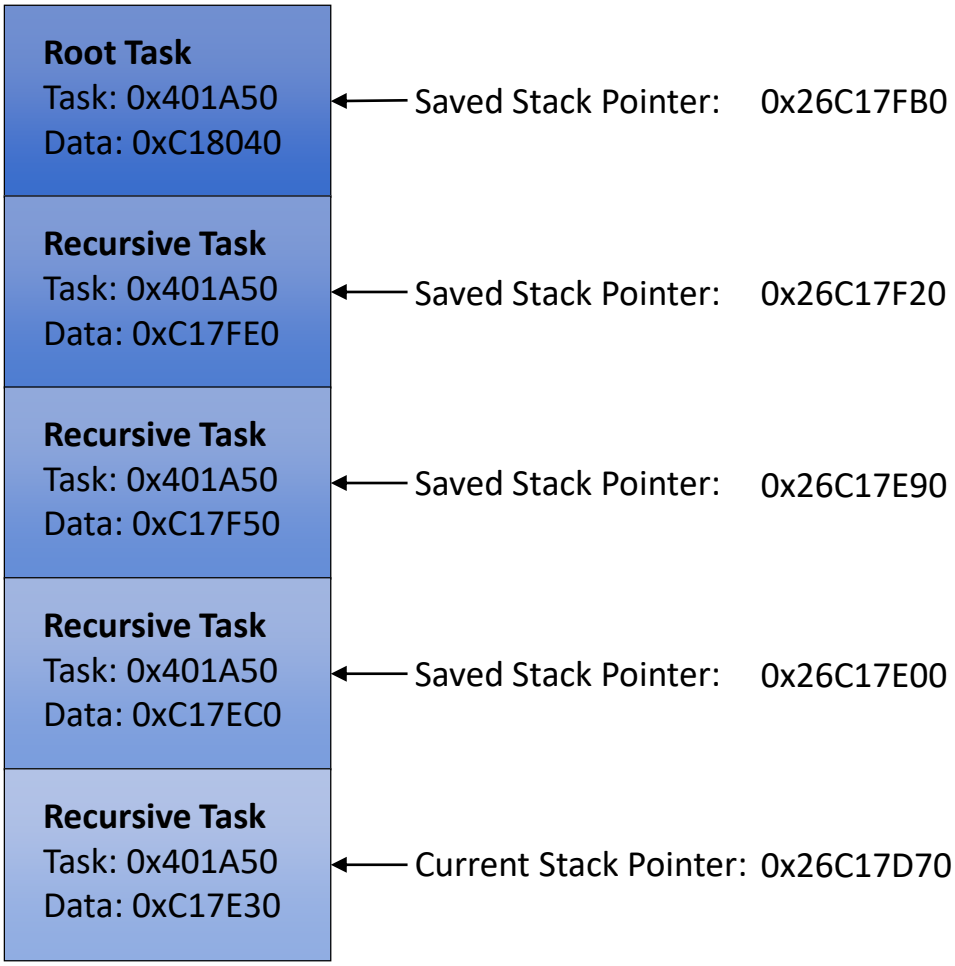

Figure 3.2: Stack with recursive fork-join tasks

returns. The handler receives a stack pointer and must check if it pertains to any model futures. The check is performed in constant time with a query to the thread module explained in Section 3.2.5. The thread module returns a model future if the stack pointer indicates a return from a fork-join task. Otherwise it returns null.

This process allows Willgrind to associate a task return with a model future. Afterwards, model maintenance and bug checking are performed.

3.2.5 Shadow Threads

Willgrind maintains a state and history of threads in the thread module. Shadowing threads is important for providing debugging information in addition to saving and retrieving stack pointers for fork-join task calls and returns. A shadow thread contains a thread ID, a history, and a stack. They are stored in a table and indexed by their guest thread ID. A callback is registered with the Valgrind core that notifies Willgrind when the client creates a new thread. This information is used to allocate and add shadow threads to the table.

Thread History

The history is an array that stores the recent model futures that a thread has created, started, completed, obtained, and freed. Just after a model future updates its state, it is added to the appropriate history. Furthermore, when the history is output, model futures are shown by their ID. This is a number generated when the model future is allocated and can be interpreted as the future's date of birth. For instance, #3 can be interpreted as a very early future in the execution and #100000 can be interpreted as late in the execution. Although this interpretation is subject to the total number of futures in the execution, it is still found to be useful for understanding a thread. Debugging using the thread history is explained more in Section 5.2.2.

Thread Stack

Stack pointers are used in Section 3.2.3 and 3.2.4 to identify fork-join task calls and returns. Moreover, each thread has a different stack and Willgrind relies on the thread module to match thread stack pointers with model futures in constant time.

To accomplish this, shadow threads use a stack data structure. When a task is invoked Willgrind passes the stack pointer and thread ID to the thread module. Next, the stack pointer is pushed onto the appropriate shadow thread's stack. At each function return,

Willgrind asks the thread module if the thread's stack pointer corresponds to a task return. To answer this query, the thread module must check if the stack pointer has previously been saved. The answer is a simple peek to compare the top of the stack with the provided stack pointer. There is no need to search the entire stack because an earlier fork-join task cannot return before the current. Thus, we use the insight that tasks must return in a LIFO order to provide constant time stack pointer lookups. If the peek and compare is positive, the stack entry is popped which contains the corresponding model future to return.

3.3 Deadlock

In a multithreaded application, deadlock describes a situation where the system does not make progress because threads are blocked forever [2]. In the case of the fork-join framework, the only way a thread can block is by entering a blocking system call. A common case of deadlock arises when system calls block for an unavailable resource, such as two threads requesting each other's locks. Another form of deadlock occurs when a system call blocks for an event that may not occur, such as a semaphore increment. In general, deadlock is difficult to detect because it is not known if/when an unblocking action may occur.

Willgrind provides accurate deadlock detection by leveraging program-specific knowledge. We use the insight that the only blocking system calls in a fork-join framework are for synchronization objects and I/O. Furthermore, we assume that threads do not deadlock on I/O requests, thus all deadlock situations involve a synchronization object. We use a heuristic in which sustained lack of progress is interpreted as deadlock. Specifically, deadlock is reported when all threads are blocked in the same system call for an extended period of time. This detection strategy appears to be reliable in our test cases because when a synchronization object cannot be acquired after a few seconds, it is likely that it will never become available.

There are two approaches for implementing this heuristic that we term eager and lazy. Eager

detection is proactive by catching deadlock before it occurs. Lazy detection is reactive by reporting deadlock after it occurs. We have experimented with both and decided that lazy detection has better tradeoffs.

3.3.1 Eager Detection

This approach does not allow threads to deadlock. To avoid deadlock, the tool must check each request to decide whether it will block [5]. For our purposes, the requests include but are not limited to the following: `pthread_mutex_lock`, `pthread_mutex_timedlock`, `pthread_barrier_wait`, `pthread_cond_wait`, `pthread_cond_timedwait`, and `sem_wait`.

Eager deadlock detection, uses function redirection to intercept the request. Upon interception, Willgrind tries the request using its try variant (e.g. `pthread_mutex_trylock`). If the try is successful, execution continues. Otherwise, Willgrind marks the thread as blocked and calls the original request. When n - 1 threads are blocked and a try is unsuccessful, Willgrind prevents the last thread from making the request and reports deadlock.

When the request returns, Willgrind marks the thread as unblocked. However, there are cases when a thread has been unblocked but just not returned from the request. For instance, a signal to unblock a condition variable could be sent but since Valgrind serializes threads, the signal will not be received until that thread runs. To avoid this false positive, a small delay and recheck must occur to allow threads to unblock before reporting deadlock.

By inspecting each request, this approach correctly detects deadlock and is not likely to produce false positives if a delay is used. The drawback is the burden of maintaining function replacements for the many different types of synchronization requests and additional overhead for checking each request.

3.3.2 Lazy Detection

This approach allows threads to deadlock and uses a watchdog to periodically check if deadlock is present. The watchdog is an observer thread that inspects the thread states once per second. If all threads are in a system call, the watchdog begins to bark. At this point, the watchdog records the stack trace of each thread, sleeps for one second, and observes the new stack traces. If at least one thread has a different stack trace, the block is diagnosed as benign and the watchdog is reset. Otherwise, the watchdog repeats the stack trace check three more times. If each thread maintains the same stack trace for three consecutive cycles deadlock is reported.

It is possible that a thread pushes and pops its calling stack resulting in forward progress with the same stack trace. However, this is unlikely and it is even more unlikely that all threads perform this behavior during the exact window when the watchdog is not looking.

This deadlock detection is simple and easy to maintain. Its drawback is the watchdog must be spawned as a client thread and excluded in bug reports.

Watchdog

Valgrind is just a single thread that runs client threads one at a time. Client threads must hold the big lock before running and drop the big lock before entering a blocking system call. Thus, when all threads are blocked the big lock is idle. As a corollary, if the client is allowed to enter deadlock, Valgrind will also enter deadlock.

The watchdog thread is necessary to keep the client and Valgrind alive in the case of deadlock. However, the watchdog must not produce errors, such as a segmentation fault, as these would inaccurately be charged as errors to the client. We use the dynamic linker to inject the watchdog into the client and the gcc builtin `__attribute__((constructor))` to run it. Additionally, the watchdog uses client requests each second to notify the tool that it is time to check for deadlock.

3.4 Web Interface

Visualizations have proven to be effective at speeding up the debugging process [25, 30]. Even several Valgrind tools have their own GUIs to provide ease of use. In Willgrind, we report our bugs through a friendly web interface shown in Figure 3.3.

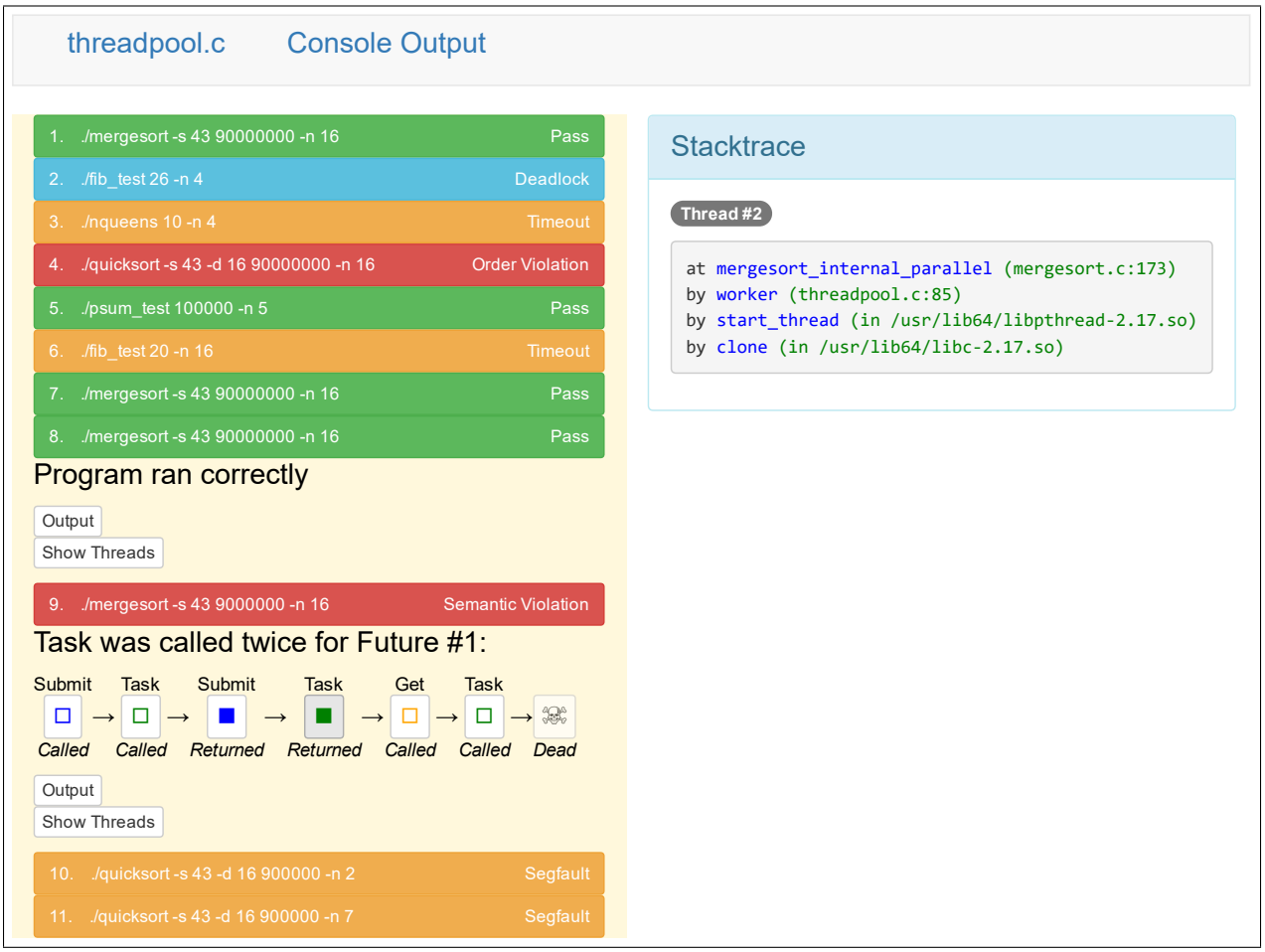

Figure 3.3: Willgrind web interface

At the top, a user may view the submitted source code and also any output generated by the program. On the left, a list of results is presented in an accordion style. In this case, there were eleven tests run. Additionally, colors are used to represent different categories of results. When a bug is reported, the user may view specific stack traces in the window on the right.

The interactivity of the interface is designed to promote investigative debugging. We have seen students leverage the variety of debugging information in creative ways to fix their bugs. Although more complicated than command line output, the web interface displays information in a way that is easier to read.

Chapter 4

WillgrindPlus

Willgrind can only detect bugs that it observes in a given execution and makes no guarantees about other executions. To address this problem, we created WillgrindPlus, a separate tool that extends Willgrind. It has all the features of the original tool but also provides happens-before based checking. This is accomplished by utilizing Helgrind's happens-before engine. Besides the new happens-before checking, WillgrindPlus is nearly identical to the original tool so we established a shared code base to avoid code clones. Figure 4.1 shows the layers of the two tools.

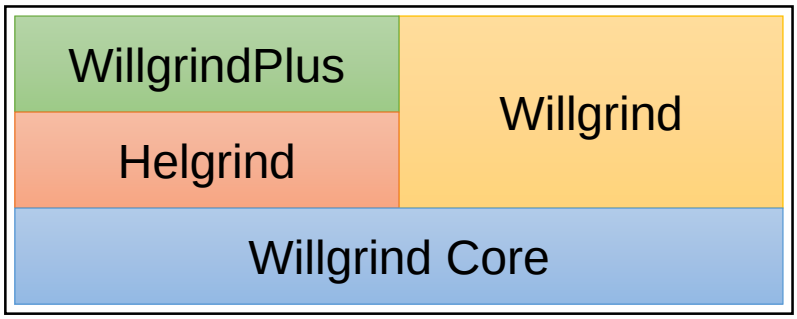

Figure 4.1: WillgrindPlus layers

WillgrindPlus still performs the same bug checking at fork-join framework events. When the model future state machine is violated, a bug is reported and the client is terminated. Oth-

erwise, happens-before based checking is performed to identify potential ordering violations from Table 3.3. We know these violations did not manifest themselves in the execution because they would have been caught in the first check. Thus, happens-before violations never indicate an actual failure in the execution but rather a latent failure. Semantic violations and deadlock are not included in the happens-before checking.

4.1 Motivation

An incorrect multithreaded program might execute correctly hundreds of times before a subtle synchronization bug appears [13]. Students have expressed frustration with Willgrind, having to run it several times to detect a bug. This is because the bug would only manifest itself for certain thread interleavings. Although, Willgrind is very useful when it detects a bug, it does nothing when a bug is hidden by the scheduling. Therefore, Willgrind's detection is limited by the scheduler.

To solve this problem, WillgrindPlus leverages happens-before based checking to augment detection. With improved detection, it is possible to detect dormant faults that could manifest themselves in a different execution. Furthermore, students only need to run the tool a minimal number of times to detect dormant bugs.

4.2 Vector Clocks

Einstein postulated that time is a relative concept. Two events A and B can be ordered A then B in one reference frame and may appear to be ordered B then A in another reference frame. This concept of time also applies to computing, in the sense that using a wall clock to establish the order of events in a system can be inaccurate. System time usually varies between two computers and synchronizing their physical clocks is extremely difficult. Lamport [32] discovered that physical clocks are not necessary to synchronize computers in

a distributed system and instead logical clocks are sufficient. Logical clocks do not measure time in seconds, but instead, use the order of events in a system to measure time. A distributed system can synchronize without clock skewing using the following insights:

- Processes do not need to synchronize if they do not interact because their events are not observable to each other.
- Processes do not need to agree on the actual time, just the order in which events occur.

Processes interact by sending and receiving messages. To synchronize, they must also send and receive logical clocks during the interactions. As a result of synchronizing, the system can determine which events happened before others establishing happens-before relationships, denoted by $\xrightarrow{\text{HB}}$. Additionally, if two events do not have a happens-before relationship, they are said to be concurrent, denoted by $||$. Lamport defined the happens-before relationship on the set of events in a system with three conditions:

1. If a and b are events in the same process and a occurred before b, then $a \xrightarrow{\text{HB}} b$
2. If a is a send event by one process and b is the corresponding receive event by another process, then $a \xrightarrow{\text{HB}} b$
3. If $a \xrightarrow{\text{HB}} b$ and $b \xrightarrow{\text{HB}} c$, then $a \xrightarrow{\text{HB}} c$

Vector clocks extend logical clocks by making it easier to compare happens-before relationships between events. Instead of one logical clock per process, each process maintains a vector timestamp (VTS). The vector's components are the logical clocks of all the processes. Figure 4.2 shows an example of vector clocks for three processes. The letters are events and the green vectors are the VTSs. At each event in a process' timeline, its own VTS component is incremented. During a send, a process shares its VTS. The receiver synchronizes by replacing any of its VTS components that are older than those in the received VTS.

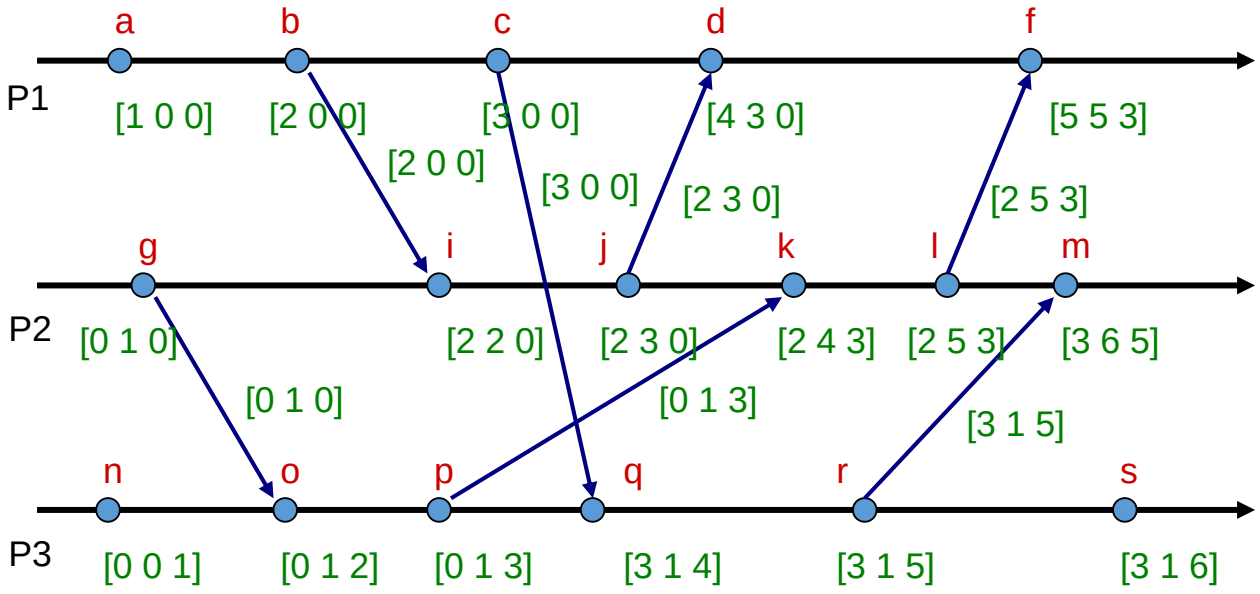

Figure 4.2: Vector clocks

By following this protocol during process communication, we can easily check if two events have a happens-before relationship. This is accomplished with a simple component-wise less than or equal comparison of the VTSs. For example, $c \xrightarrow{\text{HB}} m$ because $[3\ 0\ 0] \leq [3\ 6\ 5]$. However, the comparison does not hold for k and r. Therefore, there is no happens-before relationship between these events and $k \mid\mid r$. This means the order in which k and r occur could be observed differently in another execution.

4.3 Helgrind's Happens-Before Engine

The abstraction of processes in a distributed system can be extended to processors and even threads in a shared memory system. Since vector clocks provide happens-before relationships between events, they are particularly useful for detecting data-races in multithreaded applications. Identifying a missing happens-before relationship can indicate a lack of synchronization between threads.

The difference between threads and distributed systems is that threads do not send and

receive messages, instead they interact by accessing shared memory. To adapt vector clocks for threads, an engine is required to simulate process interaction. In Helgrind, the happens-before library provides this functionality. Although the library is its own module, it is tightly coupled with the entire Helgrind tool. For this reason, WillgrindPlus is built on top of Helgrind. Additionally, we have muted Helgrind's output and made small modifications to the happens-before library.

4.3.1 Vector Clocks in Helgrind

Helgrind models vector clocks at the thread level, where events of interest are memory accesses and process interactions are thread synchronizations. Moreover, thread synchronization consists of thread forks, thread joins, locks, semaphores, barriers, and condition variables. These synchronizations represent sending and receiving messages in a distributed system shown in Table 4.1. When a thread performs a send, Helgrind stores its VTS with respect to the corresponding synchronization object. Subsequently, when a thread performs a receive Helgrind retrieves the saved VTS and uses it to update the receiver's VTS. By emulating vector clocks at the thread level, happens-before relationships can be identified for certain events in an execution.

Send	**Receive**
Thread Fork	Thread Join
Mutex Unlock	Mutex Lock
Semaphore Post	Semaphore Wait
Enter Barrier	Exit Barrier
Condition Variable Signal	Condition Variable Wait

Table 4.1: Thread synchronization

4.4 WillgrindPlus Happens-Before Checking

Since Helgrind already checks happens-before relationships between memory accesses, reproducing this functionality is not necessary. Instead, we utilize the established vector clocks to check happens-before relationships of model future state transitions. This ensures proper thread synchronization exists between the state transitions. To do so, a small modification to the happens-before library was needed to provide API visibility to WillgrindPlus.

In our approach, each model future state transition is assigned a VTS in order to validate happens-before relationships between transitions. However, these transitions are not registered as a vector clock event and no new VTSs are created. Instead, we use the VTS of a recent existing event. This means that model future state transitions are recorded with the VTS of a memory access or a thread synchronization. Although this results in decreased granularity of happens-before checks, it provides convenience because complicated modifications to the library are not needed.

4.4.1 Happens-Before Relationships

From the model future state machine in Figure 3.1, seven happens-before relationships can be derived for state transitions. These are relationships that must exist in a correct fork-join framework. Additionally, because the happens-before relation is transitive, an additional 17 relationships can be defined. An adjacency matrix is used to represent the happens-before relationships and during the tool startup, we add transitive happens-before relationships using the Floyd-Warshall algorithm [17]. The final matrix is shown in Figure 4.3.

	Submit Pre	Submit Post	Task Start	Task Finish	Get Pre	Get Post	Free Pre
Submit Pre	1	0	0	0	0	0	0
Submit Post	1	1	0	0	0	0	0
Task Start	1	0	1	0	0	0	0
Task Finish	1	0	1	1	0	0	0
Get Pre	1	1	0	0	1	0	0
Get Post	1	1	1	1	1	1	0
Free Pre	1	1	1	1	1	1	1

Figure 4.3: Happens-before adjacency matrix. Column happens-before row

4.5 Ad-Hoc Synchronization

With the addition of atomic operations to the C11 standard, lock-free programming has become easier and more prevalent. Atomic operations are able to concurrently access memory without the use of locks [8]. Removing locks is beneficial for performance, among other reasons. This relatively new standard is influential to parallel programming practices and must be considered when designing multithreaded applications.

In the classroom, students are taught to surround every concurrent memory access with a lock. This establishes a critical section for mutual exclusion. However, the C11 atomic standard propagates a new coding convention that does not rely on mutual exclusion. For instance, a fork-join framework may set a done flag when a future is complete to indicate to another thread that its result can be returned. This is a 32-bit store that might become unnecessarily long if a lock is required to ensure thread safety. Declaring the flag as an atomic variable is a simple and efficient solution.

The reason that the done flag needs to be declared as an atomic variable is because modern hardware can optimize memory accesses. As a result, the order in which stores become visible

to other threads may not occur in program order [7]. This is illustrated by a common fork-join framework transaction in Figure 4.4 where one thread completes a future and another thread obtains the result. It is quite possible that the store of the done flag can be reordered with the store of the result. The effect of this reordering is that the obtaining thread can observe the done flag store first and return an incorrect result. When the done flag is declared atomic, there is no reordering and the transaction is completely safe [9].

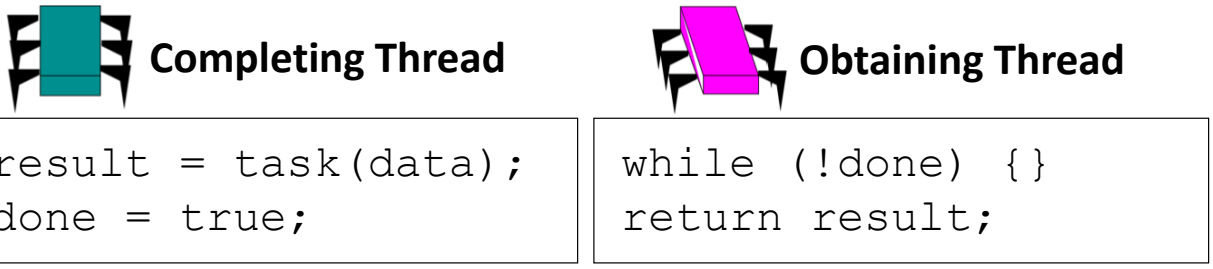

Figure 4.4: Reading and writing the done flag

However, even if the done flag is declared atomic, Helgrind still reports a data-race. This is because Helgrind does not understand the semantics of atomic variables and it relies solely on synchronization objects to establish happens-before relationships. Consequently, WillgrindPlus reports a lack of synchronization between the future completing and being obtained. This false positive can be eliminated if WillgrindPlus ignores data-races on atomic done flags.

4.5.1 Supporting the Atomic Done Flag

In order to support concurrent access to an atomic done flag in WillgrindPlus, its data-race must be suppressed. This is accomplished by disguising the memory access as if it were surrounded by a lock. Additionally, we use the assumption that a store to an atomic variable is always followed by an `mfence` instruction.

However, we do not want to suppress all data-races involving atomics variables because not all of them are benign. For our purposes, only the atomic done flag pattern has been

identified as safe. This pattern takes place when one thread writes the flag and other threads read the flag. For this reason, our heuristic keeps track of which threads have read the atomic variable and which have written to it.

During instrumentation each store is inspected to determine whether it pertains to an atomic variable. A look-ahead is performed on the instructions after the store. If an `mfence` exists, the address of the store is registered as an atomic variable. The look-ahead ends when the basic block terminates or another store is observed.

After atomic variables are identified, they are checked whenever a data-race is detected. When Helgrind reports a data-race, the memory address and conflicting threads are provided. This information is intercepted and analyzed just before the data-race is recorded. WillgrindPlus suppresses the data-race when the address has been identified as an atomic variable, the accessing thread is the writer, and the conflicting thread is the reader. At this point, the data-race is not recorded and execution continues as if it never happened. Essentially, this synthesizes a happens-before relationship shown by the red arrow in Figure 4.4. Furthermore, a transitive happens-before relationship is established between *Task Finish* and *Get Post* to prevent the false positive in WillgrindPlus.

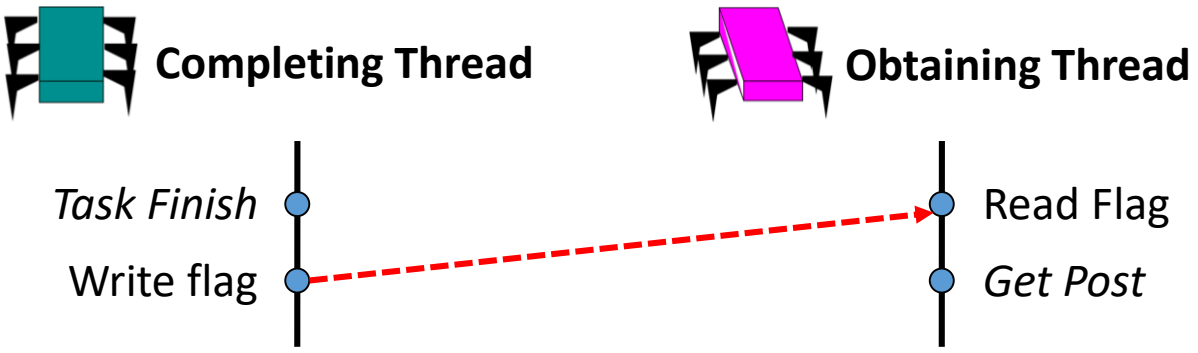

Figure 4.5: Synthesized happens-before relationship

The heuristic correctly suppresses false positives for at least two fork-join framework implementations we examined. Although, future work is needed to determine whether it is reliable. In particular, detecting atomic variables at the machine code level is not straight-

forward. Assuming that all stores to atomic variables are followed by an `mfence` may be naive because multiple memory accesses could share the same mfence. Moreover, when the default memory consistency (sequential) is not used, an `mfence` may not even exist.

Chapter 5

Evaluation

To evaluate the Willgrind and WillgrindPlus tools, we use three metrics: performance, effectiveness, and usefulness. The performance evaluation describes the cost of dynamic program analysis. The effectiveness evaluation assesses the bug detection. Finally, the usefulness evaluation investigates how important the tool is for users. As an instrument of evaluation we performed a user study on the students enrolled in Computer Systems at Virginia Tech in Spring 2017. This study was approved by the Virginia Tech Institutional Review Board (IRB 17-093).

5.1 Performance Overhead

In the performance evaluation, we compared the overhead of four tools: Nullgrind, Willgrind, WillgrindPlus, and Helgrind. All four tools run under the Valgrind framework and Nullgrind is a tool that is used to run programs without instrumentation or analysis. Additionally, we compared the performance of the tools when running on a single core and on multiple cores to investigate effects of the idle CPU state discussed in Section 2.2.4.

5.1.1 Processor Affinity Optimization

Recall that the execution of threads in Valgrind is serialized. As a result, threads block for extended periods of time when waiting to run. If each thread is assigned its own core, the cores will become idle while the threads are blocked. Eventually, when a thread unblocks, extra overhead is created for its core to exit the idle state. To avoid this performance degradation, the processor affinity of Valgrind should be set for all threads to run on a single core. In our case, we found that using the Linux command `taskset` on the Valgrind process greatly improves performance.

5.1.2 Benchmarks

Willgrind and WillgrindPlus provide a test suite for users. The suite is comprised of a medley of familiar divide-and-conquer algorithms such as merge sort, fibonacci, and queens puzzle. Furthermore, each algorithm is assessed with various sizes and thread counts. In our benchmarks, we ran the test suite on a bug-free fork-join framework implementation. Figure 5.1 shows the performance of the test suite when run natively and under the four tools.

To analyze the performance, Nullgrind is used as a measuring stick because it is the fastest possible tool. We observe Willgrind's overhead is minimal compared to Nullgrind. However, WillgrindPlus experiences a slowdown by a factor of 16. This emanates from the fact that WillgrindPlus is built on top of Helgrind and thus limited by the performance of Helgrind. The shortcomings in Helgrind's performance are due to the fact that it checks all memory accesses (including on the stack) made by the client program for data-races.

These results indicate that the overhead in Willgrind is primarily dominated by the Valgrind recompilation process. Tool developers should consider the cost of this out-of-the-box overhead when choosing Valgrind as an instrumentation framework. In regards to WillgrindPlus, the overhead is primarily dominated by Helgrind maintaining vector clocks for data-race de-

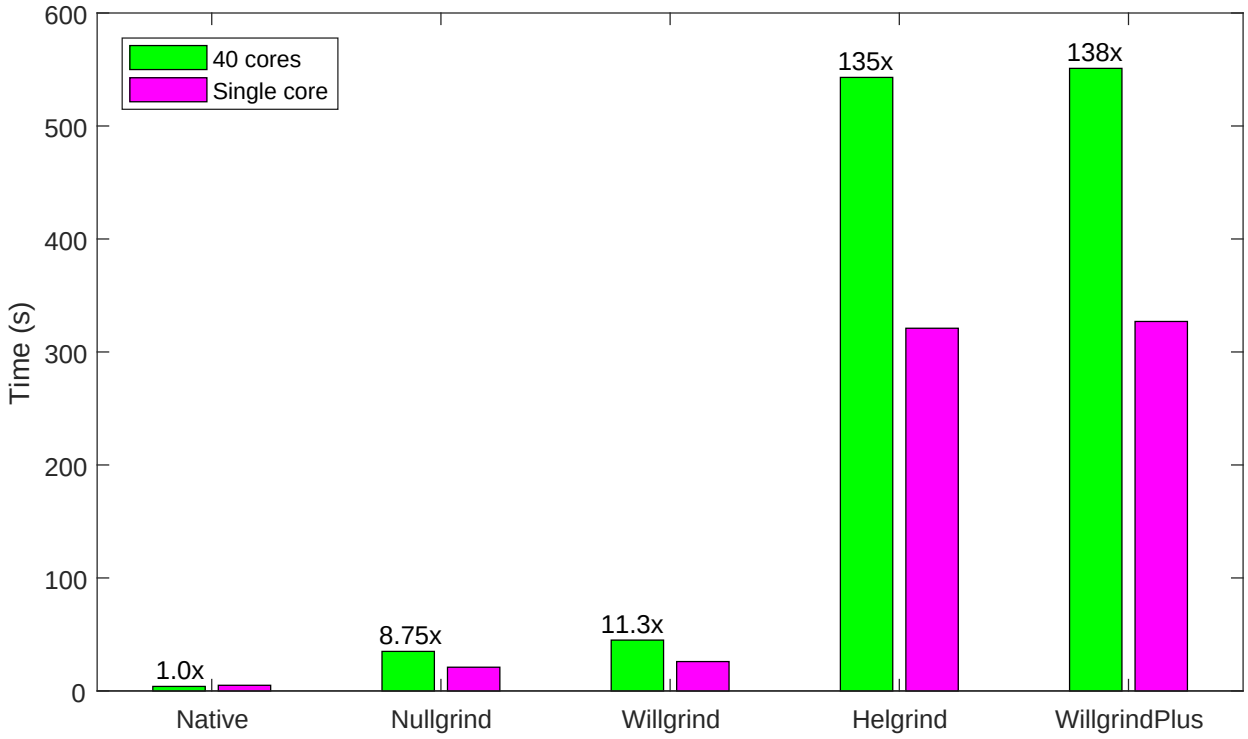

Figure 5.1: Performance of various tools on the Willgrind test suite

tection. Since WillgrindPlus does not check for data-races, a performance improvement is possible by modifying Helgrind to remove unnecessary vector clocks.

Additionally, we applied the processor affinity optimization to run the test suite on a single core. We observed roughly a 40 percent speedup for all the tools, demonstrating the high latency to enter and exit the idle CPU state [46]. Furthermore, the native execution performed 25 percent worse in single core affinity. This is expected, as the native execution can experience parallelism on multiple cores and Valgrind tools cannot.

5.2 Effectiveness

In the effectiveness evaluation, we analyze the ability of Willgrind and WillgrindPlus to detect bugs. A common approach to evaluate bug detection is to assess the tool on a

collection of buggy applications [36]. In our case, we collected the tool uses by students. With approval from the Virginia Tech Institutional Review Board (IRB #17-093), we captured code and tool results from 85 students.

5.2.1 Student Submissions

In Spring 2017, we advertised Willgrind and WillgrindPlus to students in Computer Systems as experimental debugging tools to assist them on their fork-join framework. Students were able to run the tools on their implementations and view the results through a cloud-based portal. The portal ensured confidentiality with a secure login. After the fact, personal information was stripped from the submissions to protect student anonymity from researchers.

Willgrind was used 1,851 times and WillgrindPlus was used 271 times to contribute to our collection of 2,122 unique fork-join frameworks. Although many submissions were partially complete, there were also a number of complete and bug-free fork-join frameworks. With such a large number of submissions, we observed a plethora of different outcomes. Since a single submission can be run with multiple tests, it is possible for one submission to produce multiple outcomes.

Pass This outcome indicates that the fork-join framework produced the correct solution for a test case and no bugs were detected. 42 percent of submissions produced a passing outcome.

Bug This outcome indicates an ordering or semantic violation from Table 3.3 manifested itself in the execution. 10 percent of submissions produced this outcome. The two most frequent occurrences were computing the task more than once (Bug 5) and not computing the task at all (Bug 4).

HB This outcome indicates one of the happens-before relationships from Figure 4.3 was missing. No actual failure occurred but improper synchronization was observed.

Deadlock This outcome indicates the fork-join framework entered deadlock. Users are informed that all threads are blocked and this outcome does not indicate lifelock.

Sigsegv This outcome indicates the fork-join framework (not the tool) experienced a segmentation fault.

Sigabort This outcome indicates the fork-join framework aborted execution. Typically, this occurs when the solution to the test case is incorrect or an assertion fails in the fork-join framework.

Timeout This outcome occurs when the fork-join framework does not complete in a reasonable amount of time. For each test, the tool allots an appropriate amount of time. Fork-join frameworks that exceed the time limit are either lifelocked (not blocked and not making progress) or just slow.

Crash This outcome indicates the fork-join framework caused the tool to crash. Typically, this occurs when heap meta data is corrupted as a result of the client erroneously writing past the end of a heap block. Other reasons include an assertion failure in the tool or the client executing an illegal instruction. Although the client is not always at fault, tool crashes occur only in a small percentage of submissions.

Unknown This outcome was produced by only seven submissions and indicates none of the other outcomes occurred. The cause is likely an unexpected tool crash.

Figure 5.2 shows the frequency of outcomes, not including pass. An outcome is only recorded once per submission.

5.2.2 Common Bugs

Apart from detecting bugs, a truly effective tool must help users find the root cause of bugs. In the case of Helgrind, the tool can accurately detect data-races and even localize

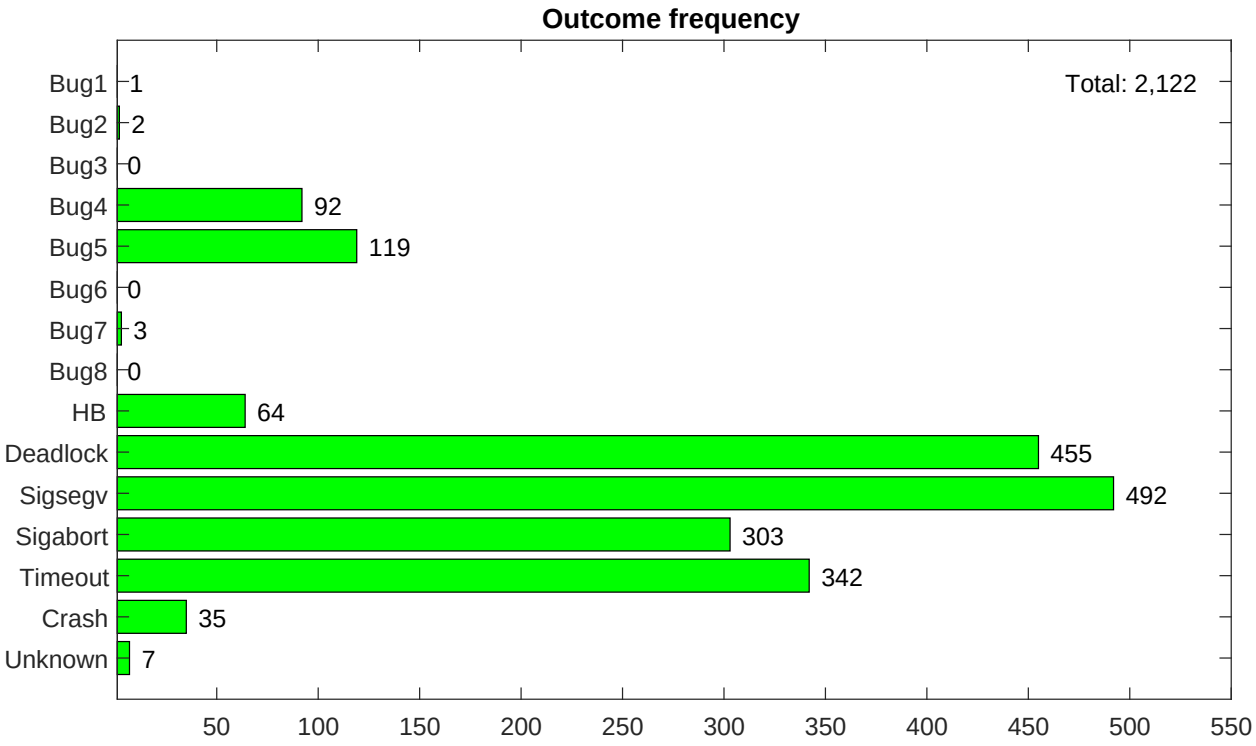

Figure 5.2: Outcomes of student submissions

the line number. However, students have expressed that Helgrind is difficult to use because the abstruse error messages do not identify the origin of bugs. To further evaluate the effectiveness of our tools, we assess their ability to provide insight on how to fix the most common bugs.

Bug 4

Bug 4 is an ordering violation that manifests itself when `future_get` returns before the task has started. A correct fork-join framework returns from `future_get` if and only if the task is complete. The root cause of this bug may be a result of an incorrect implementation of work-helping.

When a thread attempts to obtain an incomplete future, it should perform work-helping or otherwise wait for the future to complete. Work-helping is a situation where the obtaining

thread completes the future's task on behalf of another worker. Listing 9 shows an incorrect implementation of work-helping. The obtaining thread attempts to perform work-helping by completing the future on line 17. Notice that the future is not passed to `execute_task`, but rather the future's owner is. `execute_task` does not complete the future at hand, instead it completes the first future on the owner's queue. If the future at hand is not the first future on the owner's queue, its task is never executed and an incorrect result is returned. In many instances, the future at hand is the first future on the queue but this cannot be assumed in general.

```
 1  static void execute_task(void *w) {
 2      struct worker_thread *work = (struct worker_thread *)w;
 3
 4      pthread_mutex_lock(&work->workerLock);  // lock worker
 5      struct future *temp = list_entry(list_pop_front(&(work->tasks)), struct future, element);
 6      pthread_mutex_unlock(&work->workerLock);  // unlock worker
 7
 8      temp->status = EXECUTING;
 9      temp->result = temp->task(work->pool, temp->data);
10      temp->status = DONE;
11
12      sem_post(&temp->S);
13  }
14  void *future_get(struct future *future) {
15      if (future->status != DONE) {
16          // help execute other tasks
17          execute_task(future->thread);
18      }
19      return future->result;
20  }
```

Listing 9: Incorrect implementation of work-helping

The cause of this bug can be identified using the thread history feature. This feature shows recent futures that were created, started, completed and obtained at the time the bug occurred. Figure 5.3 shows the thread history that corresponds to the fork-join framework in Listing 9. We can see the only future obtained was #11 and the only future completed was #19. This indicates that the fork-join framework completed the wrong future when performing work- helping.

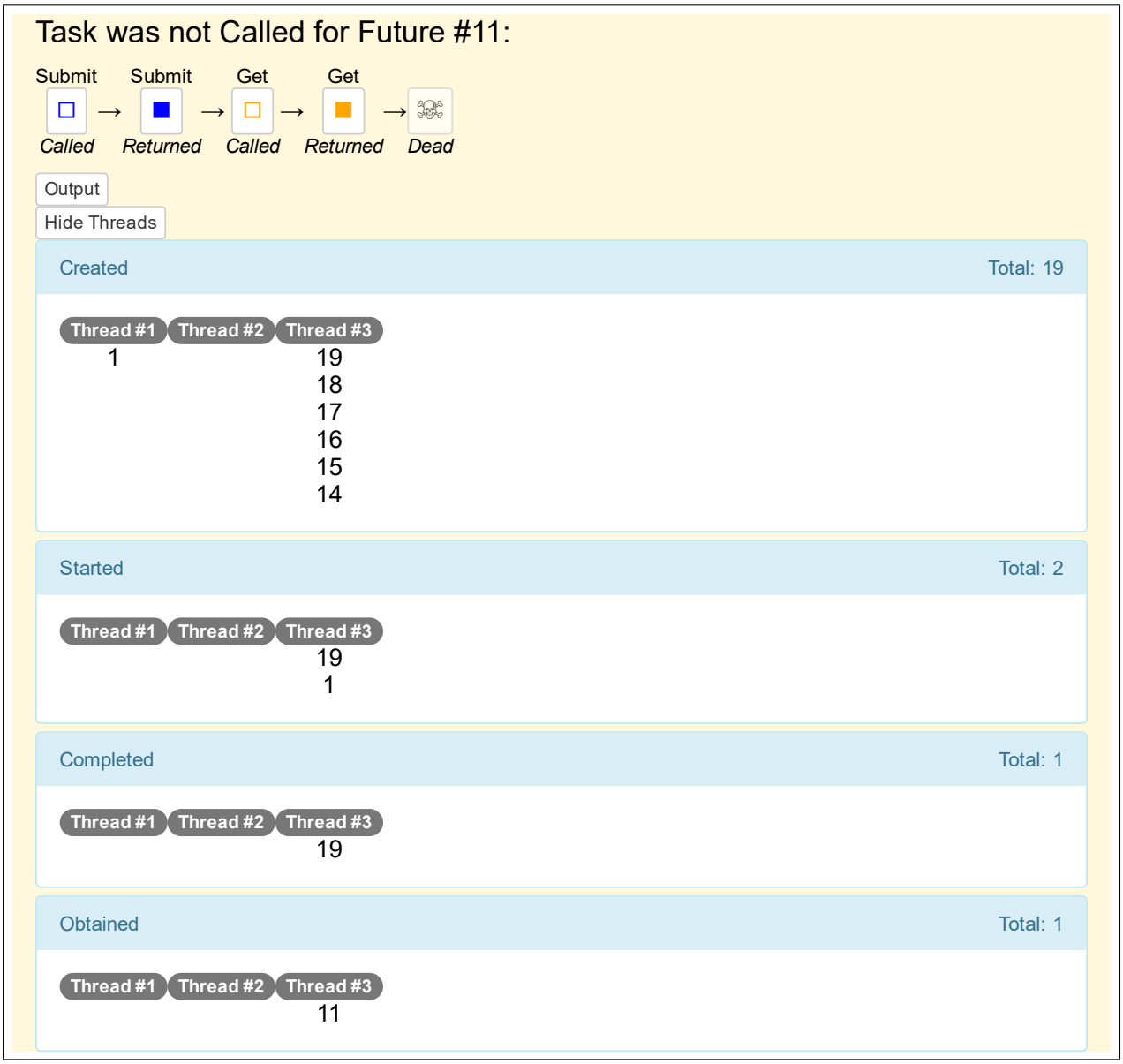

Figure 5.3: Debugging with the thread history feature

Bug 5

Bug 5 is an semantic violation that takes place when a future's task is executed multiple times. A correct fork-join framework executes each task exactly once. Although, computing a task multiple times may not produce incorrect results, it certainly degrades performance. The root cause of this bug may be a result of violating mutual exclusion.

When a thread first allocates a future it is not visible to other workers. Once a future is placed on a queue, it becomes visible to other threads and may become the target of work stealing or work helping. At this point all threads must obey mutual exclusion when operating on the future.

Listing 10 shows a future being allocated and initialized. However, before initialization is complete it is made visible to other threads on line 7. Subsequently, the started flag is initialized outside the critical section on line 10. It is quite possible a stealer could complete the task and set the started flag to 1 in between lines 7 and 10. If this occurs, the started flag gets reset to 0 and the obtaining thread recomputes the task on line 25.

```
 1  struct future* thread_pool_submit(struct thread_pool* pool, fork_join_task_t task, void* data)
 2  {
 3      struct future* future = malloc(sizeof(struct future));
 4      future->task = task;
 5      future->data = data;
 6      pthread_mutex_lock(&pool->lock);
 7      list_push_back(&pool->tasks, &future->elem);
 8      pthread_mutex_unlock(&pool->lock);
 9      sem_init(&future->done_sem, 0, 0);
10      future->started = 0;
11      pthread_mutex_init(&future->lock, NULL);
12      return future;
13  }
14  void* future_get(struct future* f)
15  {
16      pthread_mutex_lock(&f->lock);
17      if (f->started == 0) {
18          // remove from queue
19          pthread_mutex_lock(f->queue_lock);
20          list_remove(&f->elem);
21          pthread_mutex_unlock(f->queue_lock);
22
23          f->started = 1;
24          pthread_mutex_unlock(&f->lock);
25          f->value = f->task(f->pool, f->data);
26      } else {
27          pthread_mutex_unlock(&f->lock);
28          sem_wait(&f->done_sem);
29      }
30      return f->value;
31  }
```

Listing 10: Semi-initialized future made visible

A user can debug this race using Willgrind. Figure 5.4 shows the state transitions of the problematic future. We can see the task was started before `threadpool_submit` returned

and then started again after `future_get` was called. Furthermore, a user may click on the transitions to learn which threads are involved. In this case, a worker stole the future and started the task on `threadpool.c:85`. By identifying that the task was stolen and later reexecuted during `future_get` a user can understand how the data-race manifests itself and easily fix the bug.

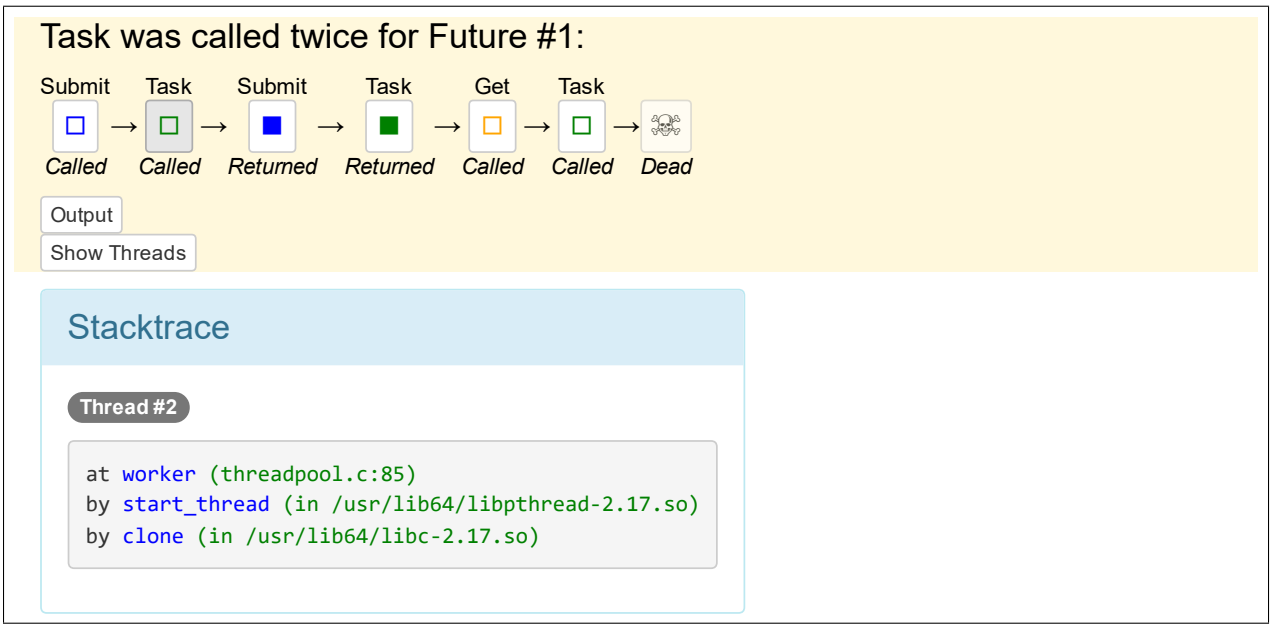

Figure 5.4: Debugging a task called twice

Deadlock

When a fork-join framework enters deadlock, Willgrind provides the exact line number where each thread is blocked. Additionally, stack traces are shown for each thread to provide a context for the user. Figure 5.5 shows an example of Willgrind's deadlock detection.

Previously, before Willgrind existed, many students received high scores on their fork-join framework even when latent deadlock existed. This is because the latent deadlock was so rare that the fork-join framework could pass the entire grading script without deadlocking. It can be difficult to identify deadlock that almost never manifests itself.

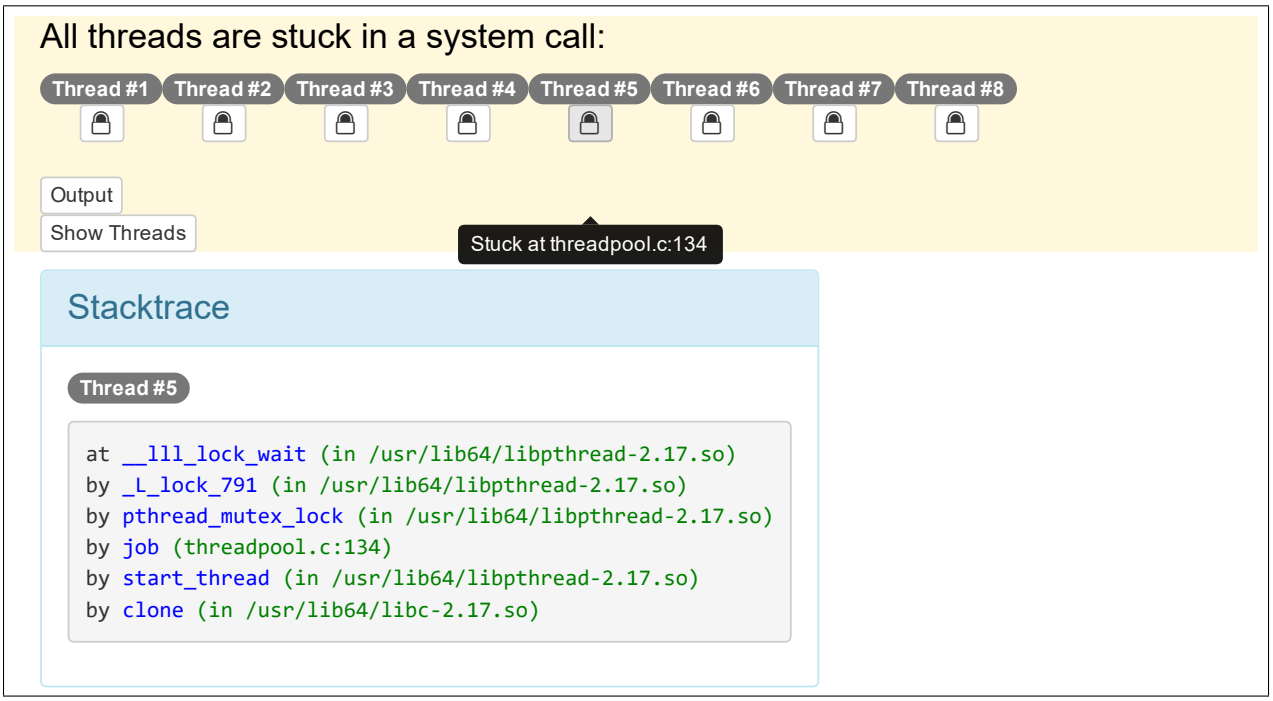

Figure 5.5: Deadlock detection

Valgrind has its own scheduler which creates extensively different thread interleavings than a native execution. The alternative scheduling is beneficial to Willgrind because it exposes some of these deadlocks that would almost never occur in a native execution. Many users of the tool complimented the deadlock detection, saying they would have never been able to catch their deadlock otherwise. Despite the praise, the deadlock detection has room for improvement as it does not report which locks are held. This is important to help identify the cause of deadlock. Without this information, users must manually reason about the lock acquisition. Nonetheless, the stack traces provide assistance in tracking down the origin of deadlock.

Segmentation Fault

Segmentation faults were the most common failure reported. Similar to deadlock, the exact line number of the segmentation fault is provided. Additionally, the alternative scheduling

is beneficial to expose rare occurrences. However, one complaint is that the alternative scheduling can have the opposite effect by hiding segmentation faults that would occur in a native execution. Nonetheless, users agreed the localization of segmentation faults is one of their favorite features. Figure 5.6 shows a null pointer dereference at line number 237.

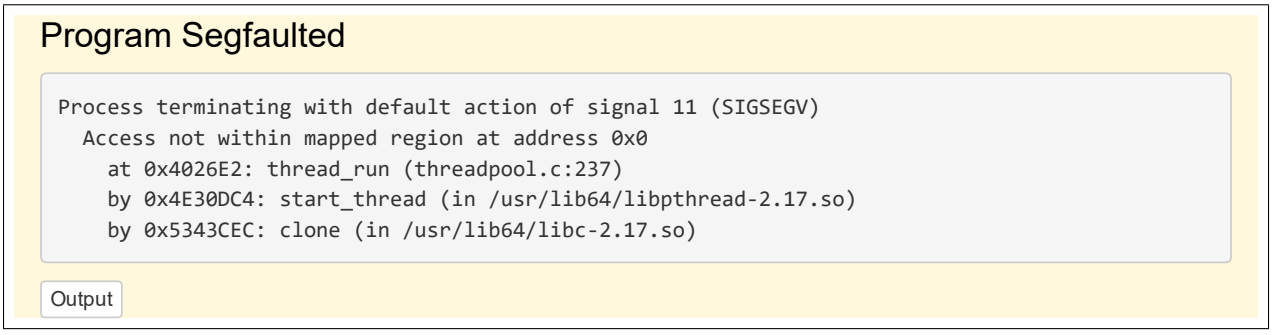

Figure 5.6: Segmentation fault localization

Happens-Before Violations

Even though WillgrindPlus checks for 24 different required happens-before relationships, 17 of them are transitive relationships. Furthermore, only three of these relationships were violated in the 271 WillgrindPlus uses:

Submit Pre || ***Task Start*** This violation is a missing happens-before relationship between the *Submit Pre* and *Task Start* events and was found in 20 submissions. There is no future, data, or task at the time of *Submit Pre* and thus, nothing to synchronize. Nonetheless, *Submit Pre* is the closest event to lack of synchronization that caused the missing happens-before relationship. Therefore, this indicates the fork-join framework is vulnerable to lack of mutual exclusion during `threadpool_submit`. A likely cause is the example from Section 5.2.2 where the future is exposed to other threads before initialization is complete.

Task Finish || ***Get Post*** This violation is a missing happens-before relationship between the *Task Finish* and *Get Post* transitions and was found in 42 submissions. This indi-

cates the fork-join framework is vulnerable to `future_get` returning an incomplete result. Typically, this violation is produced from a data-race on the future's done flag.

Task Start || ***Get Post*** This was the only transitive violation detected and was found in two submissions both from the same student. In this case the violation was caused by a combination of incorrect work-stealing and improper use of the done flag.

```
1  static void execute_future(struct future *f) {
2      pthread_mutex_lock(&f->executing_lock);
3      if (f->done == 0) {
4          f->output = f->function(f->thread_pool, f->input);
5          f->done = 1;
6      }
7      pthread_mutex_unlock(&f->executing_lock);
8  }
9  void *future_get(struct future *f) {
10     struct queue *q = f->queue;
11     while (f->done != 1) {
12         if (q != NULL) {
13             pthread_mutex_lock(&q->lock);
14         }
15         // Check if the queue has changed
16         if (q == f->queue) {
17             // Remove the future from the queue
18             if (q != NULL) {
19                 list_remove(&f->elem);
20                 f->queue = NULL;
21                 pthread_mutex_unlock(&q->lock);
22             }
23             execute_future(f);
24             break;
25         }
26         if (q != NULL) {
27             pthread_mutex_unlock(&q->lock);
28         }
29         // Set the last seen queue
30         q = f->queue;
31     }
32     return f->output;
33 }
```

Listing 11: Race on done flag

The most common happens-before violation, *Task Finish* || *Get Post*, signifies the potential for a fork-join framework to return from `future_get` before the task is complete. Listing 11

shows an example of this bug caused by the future's done flag being read without a lock on line 11. Although avoiding the lock boosts performance, it can cause `future_get` to return an incorrect value if the done flag is read out of order. However, an actual bug for this implementation has not been observed in over 10,000 executions. Furthermore, we believe this bug will never manifest itself on the x86 architecture because of certain memory order guarantees [28]. The fact that we cannot observe this bug at run-time advocates the need for happens-before checking because it is possible that a different compiler or architecture can induce this bug.

5.2.3 False Positives & Negatives

All program analysis tools must carefully consider false positives and negatives. False positives diminish the credibility of a tool and false negatives decrease the effectiveness. When addressing this quandary, some tools [51] use a threshold to balance results. In our case, Willgrind and WillgrindPlus cannot use a threshold for detection because the bugs they detect are absolute, meaning they either exist or they do not and there are no partially correct fork-join frameworks.

False Positives

To check for false positives in the tools, we manually inspected dozens of student submissions. This is a tedious process because understanding concurrent logic takes time. In Willgrind, we did not find any false positives. Additionally, we corrected a number of submissions to find that the bug reports went away. With that being said, it is still possible that false positives could exist in one of the thousands of submissions we did not check. However, our approach provides assurance by using conservative analysis to only flag bugs when there is proof. When a bug is observed, the exact cause is described to the user. For instance, when the model future state machine is violated, the state transitions of the faulty future

are shown.

In WillgrindPlus, there are no false positives under the assumption that all happens-before relationships are visible. However, false positives do exist in fork-join frameworks that use ad-hoc synchronization methods. This is because Helgrind can only recognize happens-before relationships created by synchronization objects. Therefore, if alternative synchronization idioms are used, such as atomic variables, WillgrindPlus will incorrectly flag a lack of synchronization.

False Negatives

In general, false negatives are harder to reason about than false positives. We have found that fork-join frameworks that contain bugs from Table 3.3 sometimes do not get reported. This is the result of certain thread interleavings that can hide the bugs. To solve this problem, we created a "run until failure" feature. This feature allows a user to perform stress testing for five minutes with a random sequence of tests. Although students have complimented this feature, it does not prevent the false negatives, it just makes them harder to hide in the scheduling.

WillgrindPlus is a more concrete solution to expose bugs hidden by the scheduling. Still, WillgrindPlus is also susceptible to false negatives. These false negatives are the result of spurious happens-before relationships and are common amongst all happens-before based checkers. In a specific execution, two events can accidentally have a transitive happens-before relationship caused by unrelated events that occur in between. As a result, WillgrindPlus can incorrectly observe a happens-before relationship that is only an artifact of the thread interleaving and not caused by actual synchronization [31].

5.2.4 Willgrind vs WillgrindPlus

Willgrind was run nearly seven times more often than WillgrindPlus. Nonetheless, both tools had a similar detection rate for each outcome, except HB. WillgrindPlus detected happens-before violations in three percent of submissions. The fact that Willgrind cannot detect these violations demonstrates the superior effectiveness of WillgrindPlus. However, students seemed to prefer Willgrind despite its inferior detection. Figure 5.7 shows how many times both tools were used by each student. Nearly all students used Willgrind more.

WillgrindPlus had mixed reception by students. On one hand, some students expressed WillgrindPlus was too slow and they were not interested in potential failures or what "could go wrong" with their fork-join framework. On the other hand, some students wanted a perfect fork-join framework and were able to fix all their bugs using WillgrindPlus. Furthermore, some students enjoyed being able to use both tools.

5.3 Usefulness

A common technique for evaluating the usefulness of software is performing a user study. In our case, we performed a user survey as a means for external validation and as an instrument to evaluate usefulness. 34 of the 85 student users voluntarily participated in a seven question survey. Figure 5.8 shows the responses to three of the questions. The first question evaluates whether the tool makes the project easier. A majority of respondents agree the tool decreased their time spent on the project. The second question evaluates how effective the tool is. A majority of respondents believe the tool definitely helped them fix at least one bug in their code. Additionally, none of the respondents said the tool definitely did not help fix a bug.

Finally, in the last question, users were asked if they experienced any false positives. This question is important because false positives are difficult to verify manually. Furthermore, it evaluates the credibility of the tool. Only two respondents believed the tool reported an

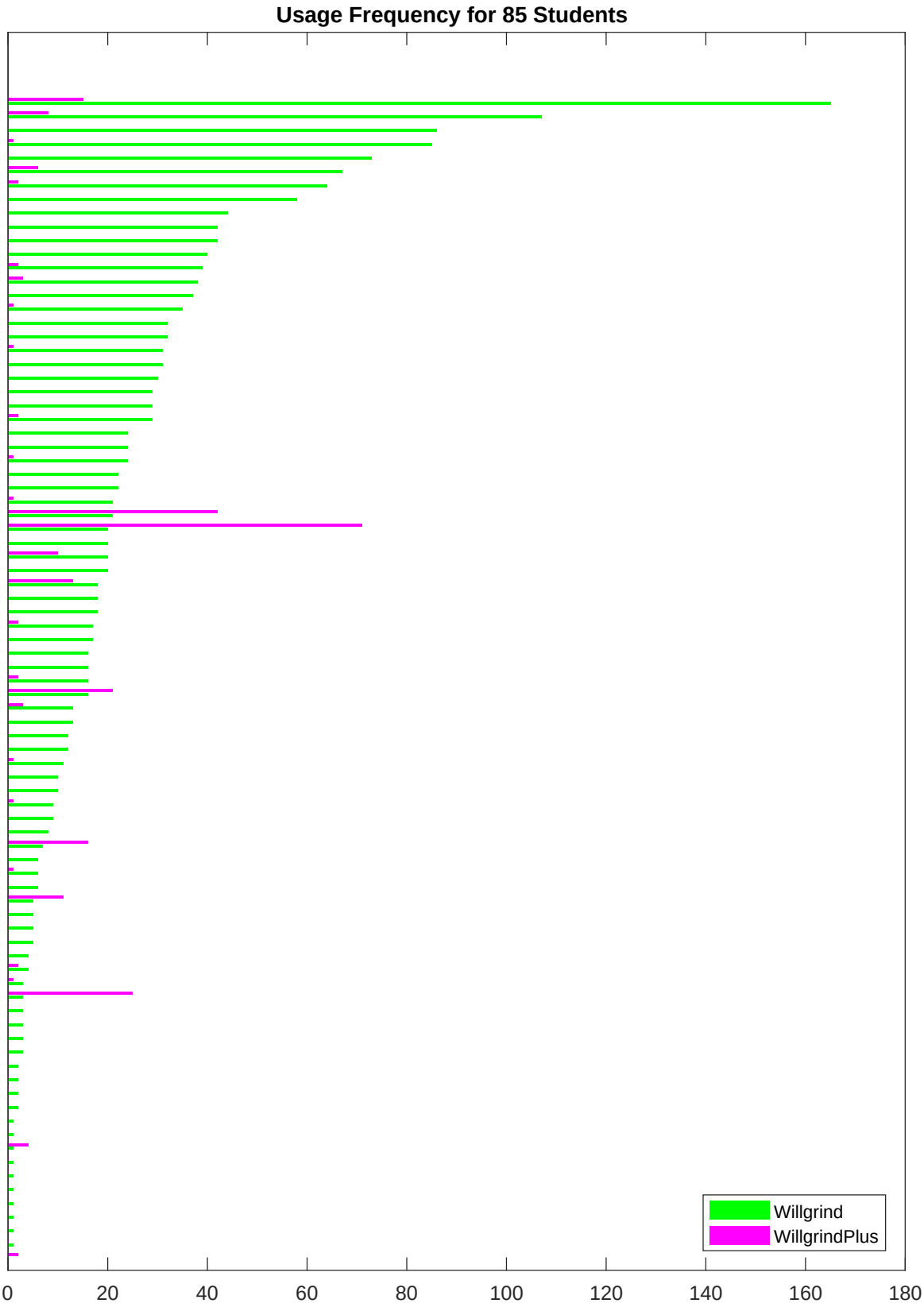

Figure 5.7: Tool usage by students, sorted by number of Willgrind uses

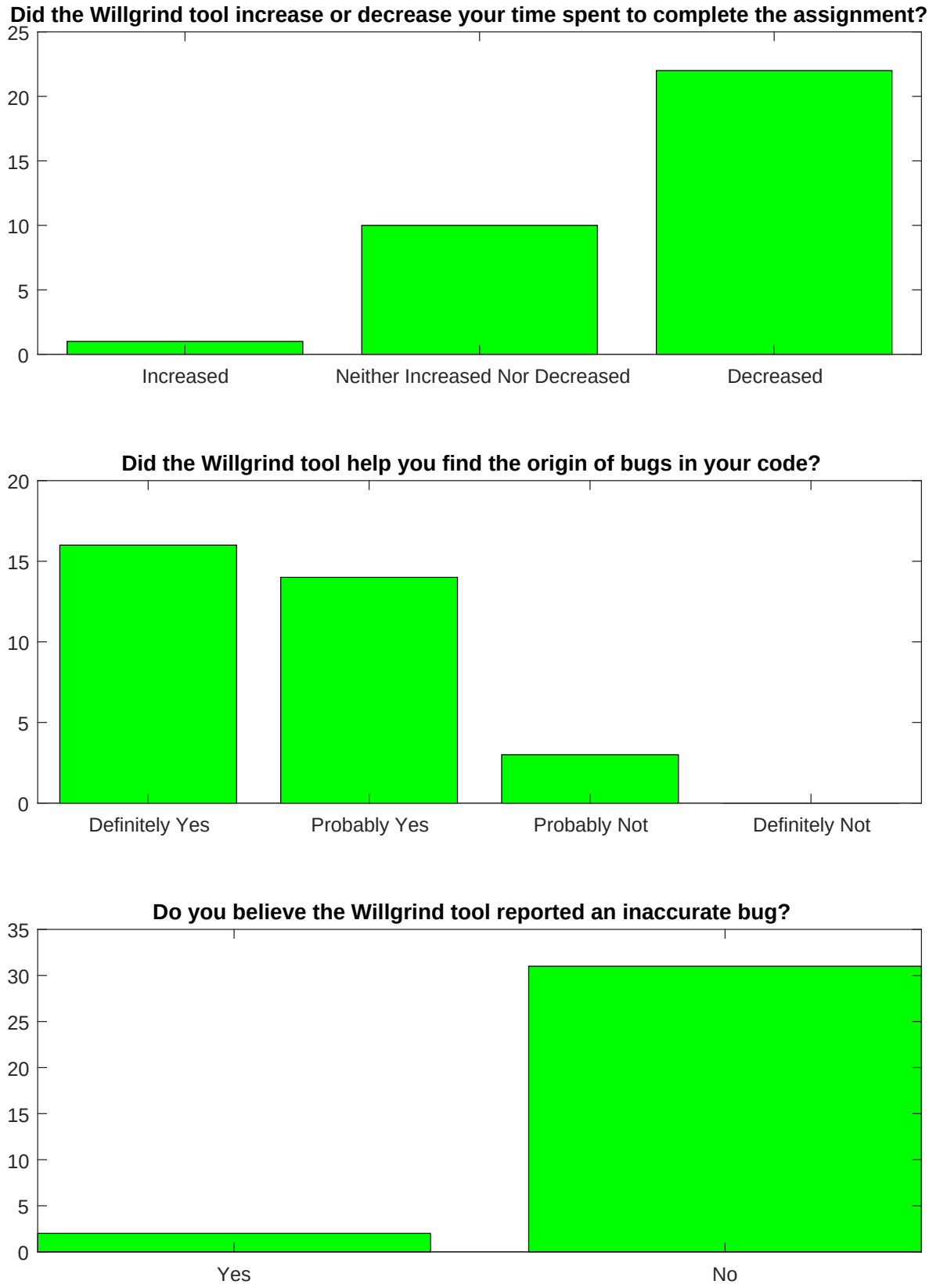

Figure 5.8: Three questions from the user survey

inaccurate bug. Since the survey was anonymous, these respondents could not be contacted. To speculate, we believe the differences between the tool and grading script could be misconstrued as a false positive. This can be due to the different scheduling when running natively and running under Valgrind.

At the end of the survey, users were asked to provide a written response on their experience. The following are some of the comments we received:

- "It was really nice to see the thread task distribution. Also it was really nice to know when we were Deadlocking versus Lifelocking versus just timing out. The GUI was really nice looking also."
- "The biggest complaint I would have is that Willgrind doesn't tell you which thread holds which lock, so deadlocks are not as easy to debug as they could be, but the tool still helps."
- "Willgrind was in the end quite useful. That said it had a nasty tendency to give you fine results for tests you knew you had just failed. Other times, when it was extremely helpful, it gave me information on my deadlocks in about 30 seconds in a much more easily readable way than GDB"
- "The willgrind tool occasionally did not catch bugs that the fjdriver tool [grading script] did catch. Otherwise, it was very useful for running specific conditions repeatedly and examining exactly what caused the program to fail."
- "WillgrindPlus aborted on a bunch of tests for us, even when we were passing the fjdriver tests [grading script]. Other than this, I really enjoyed using the Willgrind tool. Me and my partner found it especially useful that we could track the exact locks we deadlocked at. Once Willgrind showed us this, we were able to completely solve our deadlock problem in under 15 minutes."

Overall, the survey indicates that the tool is important, useful, and credible to the students.

Chapter 6

Related Work

This research is related to previous work in several domains. In this chapter, we describe different fork-join frameworks, existing race detection tools, other means for program instrumentation, and educational debugging tools.

6.1 Fork-Join Frameworks

Our program analysis tools are tailored for the special purpose of debugging the fork-join framework project. However, our approach could be generalized to create program analysis for other fork-join frameworks.

6.1.1 Cilk: A Multithreading Programming Language

The growing demand for parallel programming [54] has produced a variety of techniques to create multithreaded software. One strategy is to create a new programming language or compiler that transforms sequential programs into parallel programs [21, 12]. This describes Cilk [18], a multithreaded programming language used to improve the performance of sequential C programs.

The Cilk compiler translates a sequential program into a multithreaded fork-join program. It is especially useful for recursive programs such as divide-and-conquer algorithms. It works by analyzing source code to locate the `spawn` keyword. This keyword is used by the developer to indicate the desire for a function to become a fork-join task. At this point, the Cilk compiler adds additional statements in order to submit the function as a job for the Cilk scheduler.

The Cilk scheduler pioneered many of the strategies used in modern fork-join frameworks, notably, the work-stealing strategy. This minimizes idling and improves task distribution by allowing finished threads to steal work from busy threads. After the creators of Cilk experimented with several work-stealing tactics such as message passing, interrupting, and polling, they decided that threads should avoid direct communication and utilize the shared memory. Overall, stealing work from shared queues has become a dominant strategy in fork-join frameworks.

6.1.2 Java's Fork-Join Framework

Creating a new thread for each subproblem in a recursive algorithm can be more expensive than executing the tasks sequentially. For this reason, Lea created a fork-join framework in the `java.util.concurrent` package. It is designed similar to Cilk's work-stealing scheduler but follows an object oriented approach by using `ForkJoinPool` and `ForkJoinTask` classes. Many features of Java SE such as `java.util.Arrays.parallelSort()` utilize this framework [3].

6.1.3 Intel Thread Building Blocks

Intel Thread Building Blocks [48] (TBB) also provide a work-stealing fork-join framework useful for divide-and-conquer algorithms. Similar to Cilk, the goal of TBB is to absolve the programmer of any thread management or task distribution. However, unlike Cilk, TBB

is not a language or compiler but strictly a C++ library. Nonetheless, it provides nearly identical scalability and performance as Cilk [39], but offers the high-level abstraction of a `task_group` class. This interface allows functions to be submitted as jobs to the task scheduler. Additionally, no special compiler is needed and delegating thread management to a library increases portability.

6.2 History of Race Detectors

Data-race detectors are the primary tool for debugging fork-join frameworks. They identify when multiple threads could potentially or apparently access memory concurrently. The bugs they detect are caused by improper synchronization between threads. This section describes the history of the research in the area of race detection.

6.2.1 Eraser: A Dynamic Data-Race Detector

Lamport paved the way for data-race detection by defining happens-before relationships [32]. However, vector clocks were not always practical because they are expensive for applications with a large number of threads. This was the primary motivation behind the Eraser's lockset algorithm [52], which enforces locking discipline instead of happens-before relationships. The lockset algorithm checks that each shared memory access is protected by a lock.

This lightweight strategy improves upon heavy vector clock algorithms by increasing performance. A drawback of this approach, is that it only supports lock-based synchronization. Similar to Willgrind and WillgrindPlus, it was evaluated on multithreaded undergraduate coursework. Although Eraser had a high detection rate, many false positives existed when alternative synchronization, such as barriers and condition variables, were used. To mitigate the false positives while still achieving high performance, hybrid approaches [47] have since been introduced that combine the happens-before and lockset algorithms.

6.2.2 ThreadSanitizer

Similar to Willgrind and WillgrindPlus, ThreadSanitizer [53] is a shadow state debugging tool and is built using LLVM [33]. The primary goal of ThreadSanitizer is to provide a precise context for data-race reports, including all memory accesses and locks involved. It detects data-races using the combination of the lockset and happens-before algorithms. Unlike Eraser, ThreadSanitizer uses the original lockset of a memory access instead of the intersection locksets to decrease false positives. Furthermore, ThreadSanitizer supports additional types of synchronization, besides just locks.

6.2.3 FastTrack: Efficient and Precise Dynamic Race Detection

Previously, the consensus in the data-race community was that pure happens-before based race detection, using vector clocks, is too slow. Improvements such as Eraser and hybrid approaches are fast but imprecise because they yield false positives. FastTrack [16] offers both speed and precision by optimizing the vector clock algorithm.

Storing a vector clock for each memory access degrades performance in two ways. First, more vector clock comparisons are required. Second, cache performance is decreased, especially in applications with random accesses on large arrays. The key insight to improving vector clock performance is that an overwhelming majority of memory accesses do not need the full generality of vector clocks. FastTrack significantly reduces the number of vector clocks by grouping memory accesses together. Unnecessary vector clock allocations are avoided by dynamically identifying memory accesses that can be represented by a single vector clock. When compared to a lockset algorithm, FastTrack has slightly better performance and reports much fewer false positives.

This strategy is influential to the design of the vector clock algorithm. After abandoning a hybrid lockset algorithm, due to false positives, Helgrind (3.13) currently features an optimized vector clock algorithm similar to the one in FastTrack. This algorithm achieves

high performance by using a filter for each thread that blocks the creation of new vector clocks for certain memory accesses.

6.3 Pin: A Dynamic Binary Instrumentation Tool

Pin [38] provides an environment for tools to instrument binaries for dynamic program analysis. Along with Valgrind, Pin is one of the most popular DBI frameworks. In this section we describe the key differences, advantages, and disadvantages of using Pin.

Both Pin and Valgrind can be used to add analysis code to a client program. Additionally, both tools translate code in a just-in-time fashion at the granularity of basic blocks. The primary difference is that Pin does not use IR and instead instruments machine code directly. Since Valgrind always compiles new machine code, it can completely rewrite client programs, whereas Pintools can only add code to client programs [56]. The binary recompilation in Valgrind is advantageous for performing heavyweight instrumentation but also imposes out-of-the-box overhead which can be impractical for tools that perform minimal instrumentation. On the other hand, Pin avoids this overhead and can even attach to an existing process. However, instrumented code can materialize awkwardly when using Pin, whereas, in Valgrind, instrumented code is optimized to be integrated with the client code.

In terms of multithreading, Pin allows threads to run natively, unlike Valgrind's thread serialization. As a result, Pintools must consider thread safety. Furthermore, it is possible that multithreaded applications can experience worse performance in Pin than Valgrind because of excessive lock contention when running analysis code [53].

Overall, both tools have their advantages depending on the situation. Additionally, hybrid approaches and variations [10] are also possible. In our case, we chose Valgrind because it is favorable for heavyweight instrumentation of multithreaded applications.

6.4 LLVM: Compiler Infrastructure

LLVM [33] is a compiler infrastructure that can be used by tools to optimize and transform programs. It is similar to Valgrind and Pin in the sense that it provides a means for obtaining dynamic program analysis through instrumentation. However, unlike Valgrind and Pin, LLVM performs instrumentation at compile-time and not at run-time.

Instead of operating on binaries, LLVM leverages source code. A front-end compiler, such as Clang [1], is used to convert the source code to LLVM IR. Additionally, the front-end produces useful information that is conducive to relating the IR back to the source code. Afterwards, passes are given an opportunity to instrument the IR. Finally, once the passes are complete, LLVM compiles the IR into machine code.

Using source code for instrumentation is more straightforward than using machine code because it is easier to identify events of interest and correlate them with locations in the source code. LLVM could be a viable option for instrumenting fork-join frameworks because student source code is available. Furthermore, since LLVM performs instrumentation at compile-time, unnecessary run-time overhead can be avoided. On the other hand, LLVM does not provide useful run-time features like Valgrind's ability to inspect thread states.

6.5 Educational Debugging Tools for C

The C programming language is difficult to learn because it is designed for skilled programmers. Accordingly, many of the debugging tools for C programs are not designed for novice users. Even though novice debugging tools [11] have been created for educational languages, there is a lack of debugging tools available for students who are learning C.

When students cannot understand a debugging message they are prone to making random changes or completely rewriting regions of code [15]. These debugging habits are adverse to the education process. SeeC [14] is an educational debugging tool that detects run-time

faults in C programs. Similar to Willgrind, it instruments programs to maintain a historical state of execution and provide detailed error messages. SeeC [14] detects faults such as buffer overflows and simultaneously calling non-reentrant functions from multiple threads.

DDD [57] is another debugging tool that is used in education. It provides a graphical front-end for GDB which is helpful for students with limited debugging experience. Furthermore, the interactivity of DDD is particularly useful for students to investigate the program's execution and the state of data structures.

Chapter 7

Conclusion

7.1 Future Work

This research has demonstrated successful results in the user study. Nonetheless, there is room for improvement as indicated by the feedback. In particular, the deadlock detection only shows the line number where the threads deadlock. A truly effective tool, should not only detect deadlock but should also provide help to debug it. As an improvement to deadlock detection, Willgrind could track which locks are held to indicate the exact acquisition that caused the deadlock.

In regards to WillgrindPlus, more work is needed to support atomic variables. Currently, only the atomic done flag pattern is supported for the common case. More research is required to determine how to detect atomic variables at the machine code level, especially for different memory consistencies.

7.2 Summary

With the recent hardware design choice to increase the number of cores instead of increasing clock speed, the demand for parallel programming is higher than ever. As a result, educators are seeking new ways to introduce multithreading to students.

In our work, we make multithreaded programming easier for students by creating an analysis tool to provide a better understanding of bugs. Additionally, we extended the tool to provide advanced analysis for a deeper understanding. In the user study, the tools we developed were found to be important, effective, and useful. Future work can build upon this research to further assist students in understanding difficult programming paradigms. Our work shows that creating custom tools for classroom assignments can be beneficial in computer science education.

Bibliography

[1] Clang: a C language family frontend for LLVM. https://clang.llvm.org/. Accessed: June 25 2017.

[2] The Java tutorials - deadlock. https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html. Accessed: July 10 2017.

[3] The Java tutorials - fork/join. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html. Accessed: June 19 2017.

[4] Godmar Back. Project 2 - A fork-join framework. https://courses.cs.vt.edu/cs3214/spring2017/projects/threadpool-handout.pdf. Accessed: May 15 2017.

[5] Z. A. Banaszak and B. H. Krogh. Deadlock avoidance in flexible manufacturing systems with concurrently competing process flows. *IEEE Transactions on Robotics and Automation*, 6(6):724–734, Dec 1990.

[6] Florian Benz, Andreas Hildebrandt, and Sebastian Hack. A dynamic program analysis to find floating-point accuracy problems. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 453–462, New York, NY, USA, 2012. ACM.

[7] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM.

[8] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM.

[9] Hans-J. Boehm and Sarita V. Adve. You don't know jack about shared variables or memory models. *Commun. ACM*, 55(2):48–54, February 2012.

[10] Derek Bruening. Qz: Dynamorio: Dynamic instrumentation tool platform.

[11] Peter Brusilovsky. Program visualization as a debugging tool for novices. In *INTERACT '93 and CHI '93 Conference Companion on Human Factors in Computing Systems*, CHI '93, pages 29–30, New York, NY, USA, 1993. ACM.

[12] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[13] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. *SIGARCH Comput. Archit. News*, 37(1):85–96, March 2009.

[14] Matthew Heinsen Egan and Chris McDonald. Runtime error checking for novice C programmers. In *International Conference on Computer Science Education Innovation & Technology (CSEIT). Proceedings*, page 1. Global Science and Technology Forum, 2013.

[15] Sue Fitzgerald, Gary Lewandowski, Renee McCauley, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: finding, fixing and flailing, a multi-

institutional study of novice debuggers. *Computer Science Education*, 18(2):93–116, 2008.

[16] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[17] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[19] Anjana Gosain and Ganga Sharma. A survey of dynamic program analysis techniques and tools. In *FICTA (1)*, pages 113–122, 2014.

[20] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[21] Robert Griesemer, Rob Pike, Ken Thompson, et al. The Go programming language. *The Go Programming Language*, 2010.

[22] Dan Grossman and Ruth E. Anderson. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 505–510, New York, NY, USA, 2012. ACM.

[23] Pablo Halpern. Strict fork-join parallelism. *WG21 paper N*, 3409, 2012.

[24] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. ACM.

[25] Jerry J Harrow Jr. Runtime checking of multithreaded applications with visual threads. In *International SPIN Workshop on Model Checking of Software*, pages 331–342. Springer, 2000.

[26] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 168–181, New York, NY, USA, 2003. ACM.

[27] M. Herlihy, N. Shavit, and V. Luchangco. *The Art of Multiprocessor Programming*. Elsevier Science, 2017.

[28] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-062US. March 2017.

[29] Daniel Jackson and Martin Rinard. Software analysis: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 133–145, New York, NY, USA, 2000. ACM.

[30] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[31] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 406–422, New York, NY, USA, 2013. ACM.

[32] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[33] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[34] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[35] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[36] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5, 2005.

[37] Brandon Lucia and Luis Ceze. Finding concurrency bugs with context-aware communication graphs. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 553–563, New York, NY, USA, 2009. ACM.

[38] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[39] Sebastian Nanz, Scott West, and Kaue Soares da Silveira. Benchmarking usability and performance of multicore languages. *CoRR*, abs/1302.2837, 2013.

[40] Nicholas Nethercote and Julian Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[41] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 65–74, New York, NY, USA, 2007. ACM.

[42] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[43] Nicholas Nethercote and Julian Seward. Valgrind documentation. `http://valgrind.org/docs/manual/`, June 2017. Accessed: June 10 2017.

[44] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *In In Proceedings of the 12th Network and Distributed Systems Security Symposium*. Citeseer, 2005.

[45] Ben Pfaff, Anthony Romano, and Godmar Back. The Pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 453–457, New York, NY, USA, 2009. ACM.

[46] Nicolas Pitre. Teaching the scheduler about power management. `https://lwn.net/Articles/602479/`, June 2014. Accessed: July 9 2017.

[47] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM.

[48] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Series. O'Reilly Media, 2007.

[49] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-management architecture of the Intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March 2012.

[50] Radu Rugina and Martin Rinard. Automatic parallelization of divide and conquer algorithms. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 72–83, New York, NY, USA, 1999. ACM.

[51] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[52] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[53] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[54] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[55] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[56] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*, 2006.

[57] Andreas Zeller and Dorothea Lütkehaus. DDD - a free graphical front-end for unix debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.