Measuring the Performance of Code Produced with GitHub Copilot

by

Daniel Erhabor

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

© Daniel Erhabor 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

GitHub Copilot is an artificially intelligent programming assistant used by many developers. While a few studies have evaluated the security risks of using Copilot, there has not been any study to show if it aids developers in producing code with better performance. We evaluate the performance of code produced when developers use GitHub Copilot versus when they do not. To this end, we conducted a user study with 32 participants where each participant solved two C++ programming problems, one with Copilot and the other without it and measured the running time of the participants' solutions on our test data. Our results suggest that using Copilot can produce code with a significantly slower running time.

Acknowledgements

I want to thank my supervisors, Meiyappan Nagappan, Samer Al-Kiswany, my collaborator on this work, Sreeharsha Udayashankar, the participants for participating in the study, members of WASL and SWAG research groups, and other people who advised on things related to this work.

As a member of the University of Waterloo, I acknowledge that this work took place on the traditional territory of the Neutral, Anishinaabe and Haudenosaunee peoples.

Dedication

I dedicate this thesis to my friends and family.

Table of Contents

Author's Declaration **ii**

Abstract **iii**

Acknowledgements **iv**

Dedication **v**

List of Figures **ix**

List of Tables **x**

1 Introduction **1**

2 Background and Related Work **3**

3 Programming Problems Solved by Participants **5**

3.1 Problem selection 5

3.2 Problem A 6

3.3 Problem B 7

4 Model Solutions to the Problems **9**

4.1 Solution A 9

4.1.1 Level 0 9
4.1.2 Level 1 10
4.1.3 Level 2 11
4.1.4 Level 3 11
4.2 Solution B 12
4.2.1 Level 0 12
4.2.2 Level 1 13

5 Participants 15
5.1 Participant Recruitment 15
5.2 Difficulties Recruiting Professionals 15
5.3 Participant Summary 16

6 Experiment Design 19
6.1 Order of Solving the Problems 19
6.2 Session Introduction and Tutorial 19
6.3 Tasks 20
6.4 Timing 21
6.5 After the Problem 21
6.6 Brief Post-session Interview 21

7 Evaluation 22
7.1 RQ0 - Does using Copilot influence program correctness? 22
7.2 RQ1 - Is there a running time difference in code when using GitHub Copilot? 23
7.2.1 Approach 23
7.2.2 Results 24
7.2.3 Discussion 24
7.3 RQ2 - Do Copilot's suggestions sway developers to or from code with faster running time? 25

7.3.1 Approach . . . 25
7.3.2 Statement Level Optimizations & Open-coding . . . 26
7.3.3 Video Analysis . . . 28
7.3.4 Results . . . 28
7.3.5 Discussion . . . 29
7.4 RQ3 - Do characteristics of Copilot users influence the running time when it is used? . . . 36
7.4.1 Approach . . . 36
7.4.2 Results . . . 37
7.4.3 Discussion . . . 37

8 Conclusion **40**
8.1 Limitations . . . 40
8.2 Takeaways . . . 41

References **42**

APPENDICES **46**

A Full Description of the Problems Given to Participants **47**
A.1 Problem A . . . 47
A.2 Problem B . . . 52

B Screening survey **58**

C Tutorial **59**

D Programming Surveys **61**
D.1 First Programming Survey . . . 61
D.2 Second Programming Survey . . . 62

List of Figures

1.1 Overview of Methodology . . . 2

2.1 Copilot in Action . . . 4

5.1 Distribution of Participants' Developer Experience from Screening Survey in Appendix B . . . 17

5.2 Distribution of Participants' Familiarity with C++ from Screening Survey in Appendix B . . . 18

7.1 Plot of Problem Comprehension for Problem A vs B. "x" is the mean of the 6-Point Likert-Scale in Table 7.4 and Appendix D . . . 38

List of Tables

6.1 Factorial Matrix of **mode** x **problem** 19
6.2 Possible Orders of **mode** x **problem** 20

7.1 Table of Invalid Runs 23
7.2 Table of Statement-level Optimizations & remarks for Problem A 27
7.3 Table of Statement-level Optimizations & remarks for Problem B 28
7.4 Table of Input Data to Correlation Matrix 36
7.5 Correlation matrix for problem A 36
7.6 Correlation matrix for problem B 37

B.1 Screening Survey 58

D.1 First Programming Survey 62
D.2 Second Programming Survey Plus Demographics 64

Chapter 1

Introduction

Advances in natural language processing and deep learning have resulted in large language models (LLMs) that can generate code from free-form text. With this, language models like GPT-3 [27] have been extended to what Xu et al. [35] have termed Natural-Language-to-Code (NL2Code) generators. Notably, Open AI's extension of the GPT-3 language model, Codex [28], and the production-ready product derived from it, GitHub Copilot [1], are popular examples of NL2Code tools in use today. While some studies have shown that developers generally may have a positive experience using GitHub Copilot, others have shown that it could generate potentially vulnerable code. We present the first-ever evaluation of Copilot from a performance perspective in systems programming. We conducted the first user study on Copilot to evaluate the running time of the code generated when developers use it. With the results from our study, we hope to answer the following research questions:

RQ0: Does using Copilot influence program correctness?

RQ1: Is there a running time difference in code when using GitHub Copilot??

RQ2: Do Copilot's suggestions sway developers to or from code with faster running time?

RQ3: Do characteristics of Copilot users influence the running time when it is used?

To answer our research questions, we conduct a user study with 32 participants, where each participant solved two programming problems in C++, one problem was solved with

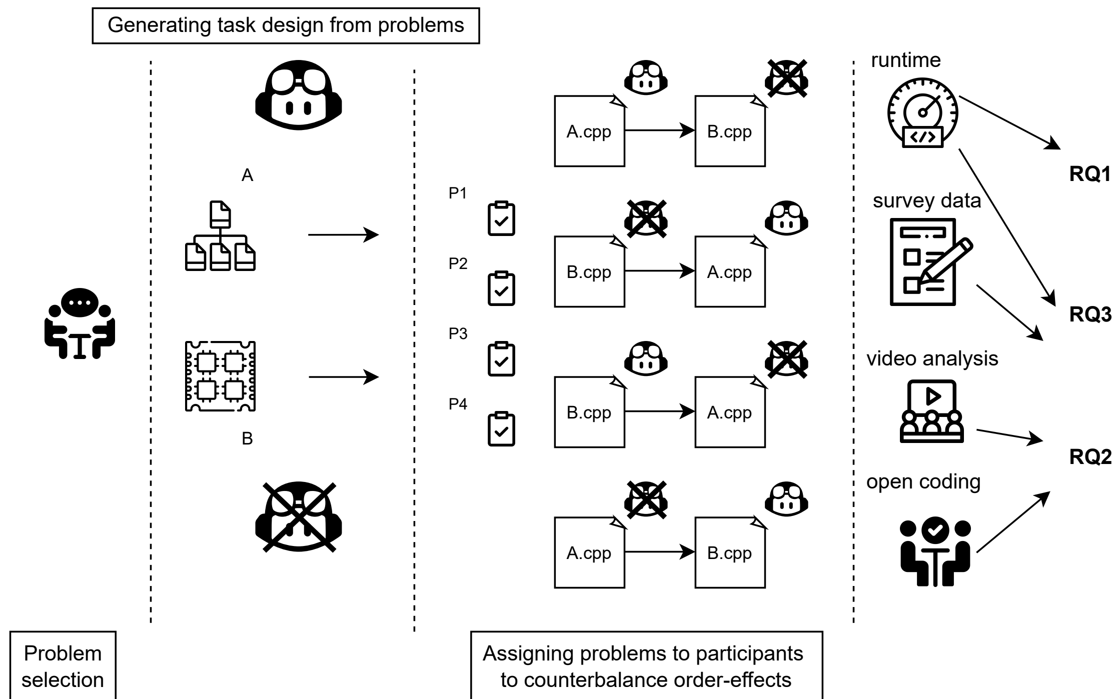

Figure 1.1: Overview of Methodology

Copilot and the other was solved without it. Our findings indicate that using Copilot resulted in code with a slower running time.

The thesis is organized in the following way: We briefly go over some background related to GitHub Copilot and some related work in Chapter 2. The process of creating the problems the participants would solve and the rationale behind choosing the problems is described in Chapter 3. Our model solutions to the problems are elaborated in Chapter 4 giving context to the problems. A summary of the participant recruitment process and the participants is described in Chapter 5. We then present the experiment design in detail in Chapter 6. An overview of the methodology can be seen in Figure 1.1. Penultimately, we analyze and discuss the experiment's results, answering our research questions in Chapter 7. Finally, in Chapter 8, we talk about the takeaways and limitations of our study and potential future directions.

Chapter 2

Background and Related Work

GitHub Copilot, the production-ready tool based on the Codex model by Open AI, can be used as a Visual Studio Code extension to suggest code snippets to users when the extension is activated. In this way, users can receive suggestions by starting to write the code or by writing comments; either way, Copilot will suggest some snippets [1]. See Figure 2.1 for an example of Copilot in action.

One of the early studies on Copilot was by Pearce et al. [31], where they wanted to understand how often suggestions from Copilot were vulnerable and the contexts that made Copilot suggest vulnerable code. To achieve this, they prompted Copilot to suggest code in scenarios where the resultant suggestions could have been vulnerable or more secure. Of the programs produced in response to the potentially vulnerable scenarios, 40% were vulnerable.

A study by Sandoval et al. [32] in collaboration with Pearce from [31] wanted to assess the security of code written by student programmers when assisted by an NL2Code assistant (OpenAI's `code-cushman-001` model) like Copilot. They conducted a between-subjects study with 58 computer science students where participants were tasked with implementing operations of a Singly-Linked List in C. Contrary to [31], their results did not show that Copilot had a conclusive impact on security.

Vaithilingam et al. [34] conducted a user study on 24 participants to understand how programmers perceive and use Copilot; they found that programmers preferred to use Copilot in their day-to-day programming tasks and found it helpful as a starting point.

```
hello.cpp 2                                         GitHub Copilot
hello.cpp > main()
1  #include <iostream>                               2
2  using namespace std;                              3  =======
3                                                    4
4                                                       Accept Solution
5  int main() {                                      5  for (int i = 0; i < 8; i++) {
6    // print hello world 8 times                    6    cout << "hello world";
7    // but on even times append a laughing emoji    7    if (i % 2 == 0) {
8                                                    8      cout << "😂";
9     Next (⌥]) Previous (⌥[) Accept (Tab) Open GitHub Copilot (^Enter)
                                                     9    }
10   for (int i = 0; i < 8; i++) {                      cout << endl;
       cout << "Hello World";                        11 }
       if (i % 2 == 0) {                             12 return 0;
         cout << "😂";                               13
       }                                             14 =======
       cout << endl;                                 15
     }                                                  Accept Solution
11                                                   16 for (int i = 0; i < 8; i++) {
12                                                   17   cout << "hello world";
                                                     18   if (i % 2 == 0) {
                                                     19     cout << "😂";
```

Figure 2.1: Copilot in Action

Chapter 3

Programming Problems Solved by Participants

Following in the same vein as Pearson et al.[31], we provided ”incomplete” code for participants to implement as a solution to a given problem. By ”incomplete”, we mean that we provided code stubs and accompanying documentation for the stubs participants were asked to implement during the study. We call the stubs ”prompts” or ”problems” and will use either of those terms interchangeably throughout this work. These prompts were provided to participants in the form of a CPP file that contained the function declaration, the unimplemented function definition that participants were expected to implement, i.e., the primary function, initialization functions and sanity checks to verify correctness. A main function was also provided as an entry point to call the initialization functions, the primary function, and the sanity checks in the appropriate order.

3.1 Problem selection

We chose two problem domains for our programming problems, file-system operations and multithreaded programming. We chose these two areas because problems in those domains tend to have a direct impact on application performance. With file I/O operations accounting for about 30% - 80% of interactions in networked file systems [29], there is a need for file system operations to be fast on storage devices [33]. Choosing a problem related to file systems reflects this demand. Additionally, since modern computing is moving towards a more parallel domain, there is a need to understand the bottlenecks of multithreaded

applications [30] and optimize accordingly. To reflect this, we chose a problem related to false sharing, a typical multi-threading optimization problem that is relatively popular [2].

We chose problems that fit the following criteria: (1) the problem must have more than one solution where each solution differs not in correctness but performance, (2) The problem should be solvable with or without Copilot assistance in 30 minutes. Problem A was in the file-system operations area, and problem B was in the multi-threading space.

3.2 Problem A

For this problem, participants were asked to read many records from three 1GB large text files and write each record to the appropriate file combination. A file combination is struct that contained a file identifier, a buffer to write the record to, and the offset for the associated file. A record is a sequence of 5000 bytes. For this problem, participants received a CPP file for prompt A and three large text files. We provide a summary of relevant declarations for more context to the problem in Listing 3.1. The full function signatures and the entirety of the CPP file with the accompanying documentation for prompt A given to participants is in Appendix A.1.

```
1  #define RECORD_SIZE 5000
2  #define NUM_RECORDS 500000
3
4  const std::vector<std::string> FILE_NAMES = {
5      "large_file_1.txt", "large_file_2.txt", "large_file_3.txt"};
6
7  struct FileCombo {
8    int fileId;
9    int offset;
10   char buffer[RECORD_SIZE + 1];
11 };
12
13 void readFileCombos(std::vector<FileCombo> &fileCombos) {
14   // YOUR CODE GOES HERE
15 }
```

Listing 3.1: Summary of Problem A

3.3 Problem B

For this problem, participants were asked to use a certain amount of threads to set all the values in a source array buffer to zero while setting all the values in a destination array buffer to a particular value. However, they were not allowed to use assignment operations, i.e., move and copy semantics were not allowed on either the source array buffer or the destination array buffer. Participants were only allowed to increment or decrement the values in the respective array buffers to solve the task. This restriction was in place because we wanted threads to repeatedly write to an item in the array and thus show the false sharing effect (depending on the implementation). We provide a summary of relevant declarations for more context to the problem in Listing 3.2. The full function signatures and the entirety of the CPP file with the accompanying documentation for prompt B given to participants are in Appendix A.2.

```cpp
const int INIT_SRC_VAL = (1 << 17);
const int SIZE = (1 << 11);
const int THREAD_COUNT = 4;

struct Item {
 private:
  int val;
  Item(const Item&);
  Item(Item&&);
  Item& operator=(const Item&);
  Item& operator=(Item&&);
 public:
  Item() { val = 0; }
  Item(int i) { val = i; }

  int get() { return val; }

  void operator++() { ++val; }
  void operator++(int) { val++; }

  void operator--() { --val; }
  void operator--(int) { val--; }
};

Item src[SIZE];
Item dst[SIZE];

void schedule() {
  // YOUR CODE GOES HERE
}
```

Listing 3.2: Summary of Problem B

Chapter 4

Model Solutions to the Problems

We created what we term "model" solutions to the problems. Because there was more than one solution to each problem, each solution we derived differed only in performance and not correctness. We itemize our solutions here and categorize them into Level 0 (L0), Level 1 (L1), Level 2 (L2), and Level 3 (L3) for problem A and Level 0 (L0) and Level 1 (L1) for problem B.

4.1 Solution A

4.1.1 Level 0

We consider a naive implementation wherein calls to `open`, `seek`, `read`, and `close` are made for each `fileCombo` in `fileCombos`. (See Listing 4.1)

```
void readFileCombos(std::vector<FileCombo> &fileCombos) {
  for(auto &fileCombo: fileCombos){
    ifstream in;
    in.open (FILE_NAMES[fileCombo.fileId], std::ios::binary);
    in.seekg(fileCombo.offset);
    in.read (fileCombo.buffer, RECORD_SIZE);
    in.close();
  }
}
```

Listing 4.1: Our Naive Level 0 (L0) Solution to Problem A

4.1.2 Level 1

One step further from the naive implementation is acknowledging that only three files are being interacted with; thus, we do not need to open and close a file for each `fileCombo` in `fileCombos`. Our optimization involves opening all the files in `FILE_NAMES` first, then processing each `fileCombo` in `fileCombos`, then closing all the files. (See Listing 4.2)

```
void readFileCombos(std::vector<FileCombo> &fileCombos) {
  std::vector<ifstream> files(FILE_NAMES.size());
  for (int i = 0; i < FILE_NAMES.size(); ++i) {
    files[i].open(FILE_NAMES[i], std::ios::binary);
  }
  for (FileCombo & fc: fileCombos) {
    files[fc.fileId].seekg(fc.offset);
    files[fc.fileId].read(fc.buffer, RECORD_SIZE);
  }
  for (ifstream & f: files) {
    f.close();
  }

}
```

Listing 4.2: Our Level 1 (L1) Solution to Problem A

4.1.3 Level 2

As a further step from L1, in this implementation, we sort the `fileCombos` by `fileId` and break ties by `offset`. This way, reading the record from an offset in a specific file will be sequential and not random. (See Listing 4.3).

```
void readFileCombos(std::vector<FileCombo> &fileCombos) {
  std::sort(fileCombos.begin(), fileCombos.end(),
            [](const FileCombo &a, const FileCombo &b) {
              if (a.fileId != b.fileId) {
                return a.fileId < b.fileId;
              }
              return a.offset < b.offset;
            });

  for (FileCombo &fc : fileCombos) {
    ifstream in;
    in.open (FILE_NAMES[fileCombo.fileId], std::ios::binary);
    in.seekg(fc.offset);
    in.read(fc.buffer, RECORD_SIZE);
    in.close();
  }
}
```

Listing 4.3: Our Level 2 (L2) Solution to Problem A

4.1.4 Level 3

A step further from L2 is a combination of the L1 optimization we did in 4.1.2 and the L2 optimization we did in 4.1.3. (See Listing 4.4).

```
void readFileCombos(std::vector<FileCombo> &fileCombos) {
  std::vector<ifstream> files(FILE_NAMES.size());
  for (int i = 0; i < FILE_NAMES.size(); ++i) {
    files[i].open(FILE_NAMES[i], std::ios::binary);
  }

  std::sort(fileCombos.begin(), fileCombos.end(),
            [](const FileCombo &a, const FileCombo &b) {
              if (a.fileId != b.fileId) {
                return a.fileId < b.fileId;
              }
              return a.offset < b.offset;
            });

  for (FileCombo & fc: fileCombos) {
    files[fc.fileId].seekg(fc.offset);
    files[fc.fileId].read(fc.buffer, RECORD_SIZE);
  }

  for (ifstream & f: files) {
    f.close();
  }

}
```

Listing 4.4: Our Level 3 (L3) Solution to Problem A

4.2 Solution B

4.2.1 Level 0

We consider a naive implementation to be a solution where each thread starts at the respective indices 0, 1, 2, and 3 (where `THREAD_COUNT` is 4) in the `src` and `dst` arrays. Each thread then decrements and increments the `Item` in `src` and `dst`, respectively. (See Listing 4.5). After processing the respective `Item`, each thread moves `THREAD_COUNT` steps until the next index, i.e., 4, 5, 6, and 7 and processes the `Item` therein. We consider this the

naive solution because false sharing is present because each thread invalidates the same 64-byte cache line when decrementing and incrementing the `Item` at `src` and `dst` arrays.

```
void work(int start) {
  for (int i = start; i < SIZE; i += THREAD_COUNT) {
    for (int j = 0; j < INIT_SRC_VAL; ++j) {
        --src[i];
        ++dst[i];
    }
  }
}

void schedule() {
  std::thread threads[THREAD_COUNT];
  for (int i = 0; i < THREAD_COUNT; ++i) {
    threads[i] = std::thread(work, i);
  }
  for (int i = 0; i < THREAD_COUNT; ++i) {
    std::threads[i].join();
  }
}
```

Listing 4.5: Our Naive Level 0 (L0) Solution to Problem B

4.2.2 Level 1

Our second optimization level is to avoid false sharing by dividing each array (`src` and `dst`) into `THREAD_COUNT` slices and assigning a single thread to process each `Item` in that slice. While we acknowledge that aligning the `Item` struct definition in Listing 3.2 to 64 bytes (the cacheline size) could be another way of avoiding false sharing, we chose not to give participants the flexibility of modifying the struct definition and thus potentially violating the time limit constraint for the problem. (See Listing 4.6)

```cpp
void work(int start, int end) {
  for (int i = start; i < end; ++i) {
    for (int j = 0; j < INIT_SRC_VAL; ++j) {
      --src[i];
      ++dst[i];
    }
  }
}

void schedule() {
  int slice = SIZE / THREAD_COUNT;
  std::thread threads[THREAD_COUNT];
  for (int i = 0; i < THREAD_COUNT; ++i) {
    threads[i] = std::thread(work, i * slice, (i + 1) * slice);
  }

  for (int i = 0; i < THREAD_COUNT; ++i) {
    threads[i].join();
  }
}
```

Listing 4.6: Our Level 1 (L1) Solution to Problem B

Chapter 5

Participants

5.1 Participant Recruitment

Participants were recruited mainly via the mailing list for computer science graduate students and snowballed to other interested participants. We focused on systems developers. We consider participants as systems developers if they had taken a systems course including but not limited to Operating Systems, Distributed Systems, or Computer Networking. We also considered individuals as system developers if they were involved in systems development professionally, with open-source contributions included.

To be eligible for the study, potential participants needed access to an internet browser and GitHub Copilot on VSCode at the time. They also must be a system developer as described above, must have had at least a few months of programming experience, and must have had some familiarity with the C++ programming language. Additionally, to be eligible, participants could not be employed by Open AI or GitHub or involved with the development of GitHub Copilot at the time.

To check if potential participants were eligible to participate, they were sent a Qualtrics screening survey after they had read and signed the consent form declaring their intent to participate. Details of the screening survey can be found in Appendix B.

5.2 Difficulties Recruiting Professionals

At the halfway point of our desired participant goal, we paused participant recruitment to analyze the preliminary data we had obtained. On looking at the snapshot of participants'

solutions to problem A, we noticed that not a single participant had implemented any of the three levels of optimizations we had considered when designing the problem. At the time, most of the participants had been graduate students with sound systems backgrounds, i.e., they were part of a research group that focused on systems. However, we decided to diversify our participant pool by including professional systems developers. The initial process of attempting to recruit professional systems developers started with contacting alums of the affiliated university who were known to be working as systems developers. Additionally, we looked for contributors to systems projects on GitHub that were primarily implemented in C++. The advanced search feature was used to find projects that contained the keywords "systems", "operating systems," or "database". We fine-grained our search to projects with a dedicated social platform where interested parties connect, i.e., Discord [4] and Internet Relay Chat (IRC) [8]. Projects such as SerenityOS[14] and SkiftOS[15] had active Discord communities; however, there was a paucity of interested potential participants in the study.

Attempts to garner interest in the study from said project contributors were met with either backlash or suggestions to reach out to other Discord communities such as the osdev (Operating Systems Development) [12] discord and the associated IRC. Upon interacting with the osdev community on the Discord and IRC platforms, there was a general unwillingness to participate in the study, with community members citing potential copyright issues with Copilot and other negative perceptions of GitHub Copilot, GitHub, and Microsoft. Thankfully, recruitment efforts paid off as a few (less than we would have liked) professionals were willing to participate in our study and thus met our desired participant goal.

5.3 Participant Summary

We recruited a total of 32 participants for the study, where 8 were professionals in systems programming or contributors to open-source systems projects. 23 were graduate students with a systems research area at the time of participating and one was a sessional lecturer but was previously a graduate student with a systems research focus. The distribution of the participants' experience is in Figure 5.1 and their familiarity with C++ is in Figure 5.2. Participants were compensated $50 for their time and the study was approved by Research Ethics Board (**REB# 44162**) at the affiliated university.

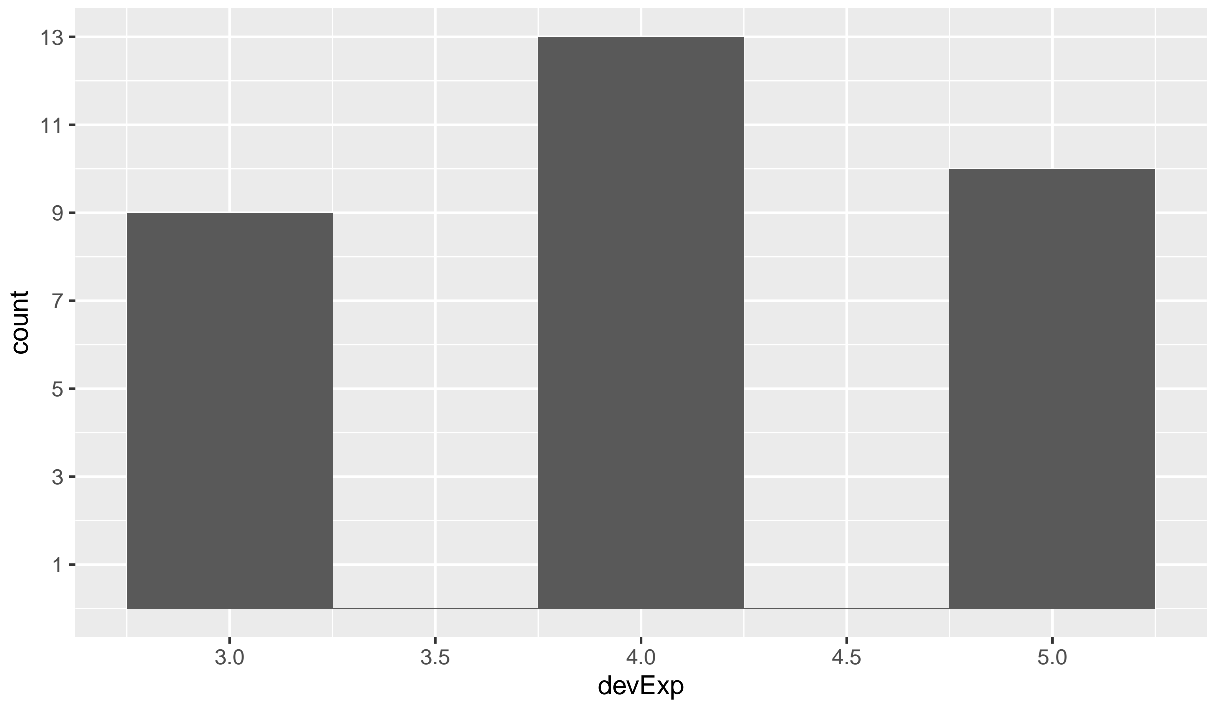

Figure 5.1: Distribution of Participants' Developer Experience from Screening Survey in Appendix B

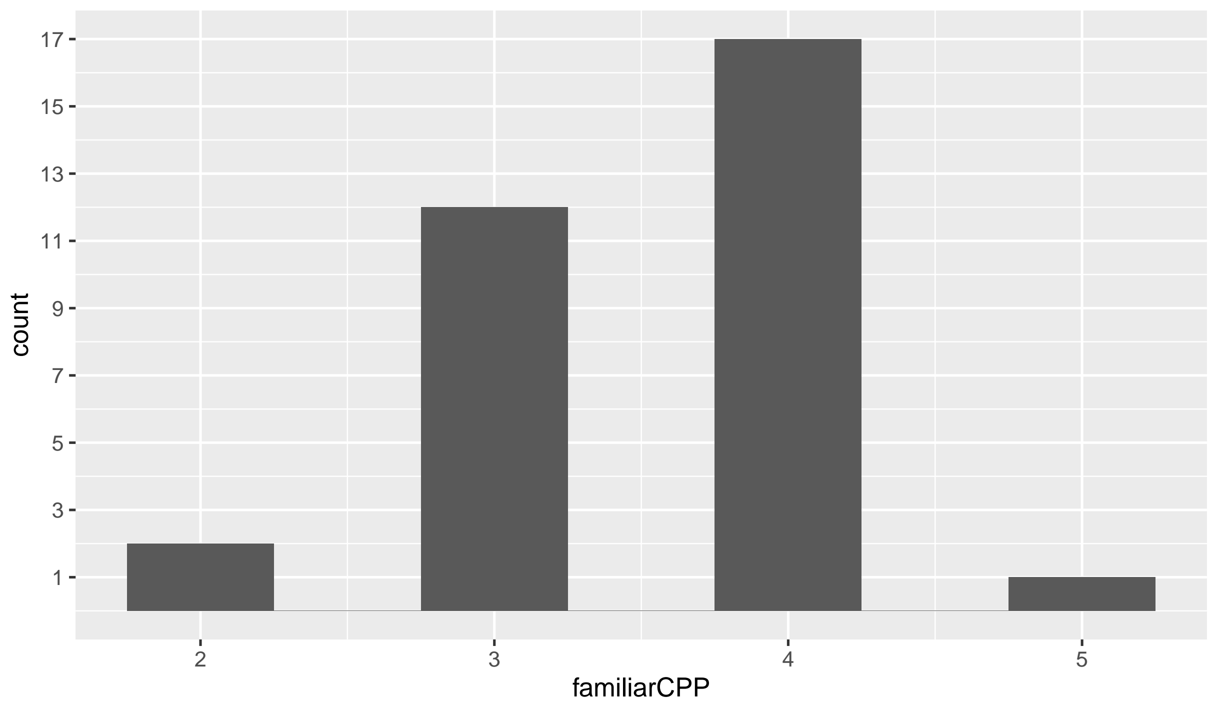

Figure 5.2: Distribution of Participants' Familiarity with C++ from Screening Survey in Appendix B

Chapter 6

Experiment Design

6.1 Order of Solving the Problems

Given our within-subjects experimental design where one participant solves one problem with Copilot and then the other problem without it, we needed to ensure that any order effects are counterbalanced across all 32 participants. To this end, we present a factorial matrix where the prompts (**A** and **B**) are the column labels, and the modes (**C** and **NC**) are the row labels which indicate using Copilot and not using Copilot respectively. (See Table 6.1). We further expand this matrix to the product of prompts and modes. Four possible orders of problem x mode are generated (See Table 6.2).

The orders in Table 6.2 enforced a requirement that our participant pool be a multiple of four. Hence, we recruited 32 participants for the study.

axes	**A**	**B**
C	C x A	C x B
NC	NC x A	NC x B

Table 6.1: Factorial Matrix of **mode** x **problem**

6.2 Session Introduction and Tutorial

The session was done remotely on an online conferencing platform. It started with the facilitator introducing the study, confirming the participant's consent to participate, and

#	first	second	Participant ID
1	C x A	NC x B	P1
2	NC x B	C x A	P2
3	NC x A	C x B	P3
4	C x B	NC x A	P4

Table 6.2: Possible Orders of **mode** x **problem**

then confirming the participant's number (the participant ID given to the participant once eligibility and consent were given before the session). The facilitator then continued by explaining the overview of participant responsibility. They then requested the participant's consent to record the audio and the screen during the session. Finally, the experimenter gave a tutorial on what was expected, from opening the problem in VSCode to using Copilot to accept, reject, and view all suggestions and zipping the edited code files. (See Appendix C for more details on the tutorial)

6.3 Tasks

Participants were given two C++ programming problems to solve within 30 minutes each. Each prompt was self-contained within a CPP file within a compressed folder, i.e., participants were given a ZIP file that contained a CPP file which contained the prompt. The ZIP file was sent to the participant via the online conferencing platform's chat feature or a Google Drive link if technical issues with the "Upload File" feature ensued.

Upon opening the unzipped folder in VSCode, it was emphasized that the contents of the CPP files should not be visible until the experimenter indicated otherwise. After verbally confirming that the folder was open in VSCode and their browser was ready, the participant was asked to share their entire screen.

The facilitator then confirmed that (1) all extensions were disabled except for the Copilot extension.[1] (2) the participant could easily switch between their browser and VSCode. At this point, it was further emphasized that the browser and other resources or references could be used in addition to GitHub Copilot.

[1] keybinding related extensions like VSCode Vim[25] and ModalEdit[11] and SSH-related extensions like Remote - SSH[23] and WSL[24] were the only exceptions allowed

6.4 Timing

Once the above requirements were met, the facilitator let the participant know they would solve the problem within 30 minutes. Moreover, the participant will be alerted when 20 minutes were left, 10 minutes were left, 5 minutes were left, and when the timer ran out. Following this, they were asked to open up the prompt (the contents of the CPP file should be visible), activate Copilot (if Copilot was to be used for this problem), and start. At this point, the experimenter started the timer.

6.5 After the Problem

Once the participant declared that they were done or if the timer ran out, the experimenter stopped the timer and instructed them to compress the entire folder and send it via the chat feature or other means (Google Drive link or email). The experimenter let the participant know they were stopping the timer before actually stopping it, so participants were aware of the finality of declaring that they were done with the programming task.

Upon sending the compressed folder containing the edited code files, participants were asked to deactivate Copilot (if activated) and close their VSCode window, browser window, and other references they had opened. The intent was to minimize any learning effects upon encountering the second problem. The participant was then sent a link to a survey to complete. Upon completing the survey, the participant could take a break before beginning the second problem.

The instructions and procedure for the second problem were the same as the first differing only in the survey at the end. The second survey contained demographic questions in addition to the initial survey questions. Details of the first and second surveys are outlined in Appendix D.

6.6 Brief Post-session Interview

Following the session, participants were asked for feedback about the study, GitHub Copilot or anything they wanted to share.

Chapter 7

Evaluation

All participants' code was run on a Linux machine running Ubuntu 20.04 with eight-core Intel Xeon D-1548 at 2.0 GHz, 64GB ECC Memory (4 x 16 GB DDR4-2133 SO-DIMMs), and 256 GB NVMe flash storage and compiled with gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1 20.04) [7]. We ran each participant's code 32 times to accommodate any slight variations in running time between each run. Additionally, the filesystem cache was cleared for each run of the 32 runs for each participant's code.

In events where the participant's code did not compile, the participant's code was not run and thus not analyzed. In events where participants' code did compile and run without error but failed the sanity checks, the running time was recorded but not used in the analysis, i.e., we only considered correct solutions before looking at the performance. Additionally, in an event where the participant's code did compile but ran with an error, the running time was not obtained and thus was not analyzed. A particular occurrence of this ensued when a participant's code produced a segmentation fault error even though it compiled successfully.

7.1 RQ0 - Does using Copilot influence program correctness?

As a precursor to our main research questions, we note that out of 32 participants where 16 attempted to solve problem A with Copilot and another 16 without it, all 16 participants' solutions passed the sanity checks when Copilot was used. However, 4 out of 16 code

snippets either did not compile (P15 and P7), ran and failed the sanity checks (P3), or ran with errors (P23) when Copilot was not used for problem A.

In problem B, however, we note that out of 32 participants where 16 attempted to solve problem B with Copilot and 16 without it, we see that 14 participants' code passed the sanity checks both when Copilot was used and when not used. The 2 "invalid" solutions where Copilot was used either did not compile (P15) or ran and failed the sanity checks (P32). The 2 "invalid" solutions where Copilot was not used, compiled but failed the sanity checks (P30 and P6). See Table 7.1 for a summary of the invalid solutions where **partID** is the anonymized participant ID, **problem** is the problem type A or B, **mode** indicates whether Copilot was used (C) or not (NC), **compiled** indicates whether the code compiled (TRUE) or not (FALSE) and **passed** indicates whether their code passed the sanity checks (TRUE) or not (FALSE). NULL, if their code did not compile or had a runtime error.

#	partID	problem	mode	compiled	passed
1	P32	B	C	TRUE	FALSE
2	P30	B	NC	TRUE	FALSE
3	P23	A	NC	TRUE	NULL
4	P15	A	NC	FALSE	NULL
5	P15	B	C	FALSE	NULL
6	P7	A	NC	FALSE	NULL
7	P6	B	NC	TRUE	FALSE
8	P3	A	NC	TRUE	FALSE

Table 7.1: Table of Invalid Runs

7.2 RQ1 - Is there a running time difference in code when using GitHub Copilot?

7.2.1 Approach

To answer this question, we compare the running time of all 32 runs of the participants' source files for problems A and B. We use the non-parametric Wilcoxon rank sum test in R [26] `wilcox_test()` to compare the running times.

7.2.2 Results

For valid solutions to problem A with Copilot (n1 = 16 x 32), the summary statistics are as follows: *mean* - **34.86 s**, *median* - 34.85 s, *min* - 33.82 s, *max* - 36.02 s. For valid solutions to problem A without Copilot (n2 = 12 x 32), the summary statistics are as follows: *mean* - **26.02 s**, *median* - 34.47 s, *min* - 4.045 s, *max* - 35.84 s. On comparing the running times of solutions to problem A for not using Copilot with using Copilot (p = **3.4e-34**), we find the results to be statistically significant. In this way, we observe that not using Copilot was about **29%** faster than using Copilot to solve problem A when comparing the mean running time.

For further context into the running time, we also ran our L1, L2, and L3 solutions for problem A for 32 runs alongside the participants' source for a fairer evaluation. Our model L1 solution had a mean of **30.59 s** and is **13%** faster and **16%** slower than solutions with Copilot assistance and without Copilot assistance, respectively. Our model L2 solution averages **7.565 s** and is **129%** and **110%** faster than participants' solutions when using Copilot and not Copilot, respectively. Our model L3 solution had a mean of **5.288 s** and is **147%** and **132%** faster than participants' solutions when using Copilot and not Copilot, respectively.

Similarly, for valid solutions to problem B with Copilot (n1 = 14 x 32), the summary statistics are as follows: *mean* - **1898 ms**, *median* - 945.4 ms, *min* - 612.1 ms, *max* - 7356 ms. For valid solutions to problem B without Copilot (n2 = 14 x 32), the summary statistics are as follows: *mean* - **1628 ms**, *median* - 943.9 ms, *min* - 494.9 ms, *max* - 6761 ms. On comparing the running times of solutions to problem B when Copilot was used versus when it was not used ($p = 0.000058$), we also find the results to be statistically significant. Further, we also observe that not using Copilot for problem B was about **15%** faster than using Copilot.

We provide a similar context into the running time by running our L1 solutions for problem B for 32 runs alongside the participants' source for a fairer evaluation. Our model L1 solution averages at **581.4 ms** and consequently **106%** and **95%** faster than participants' solutions when using Copilot and not using Copilot respectively.

7.2.3 Discussion

Our results suggest that developers may benefit from Copilot-unaided code. We give further context to these results by highlighting some participants' Copilot-unaided solutions

whose mean running times were close to or better than the model solutions highlighted in Chapter 4.1 and Chapter 4.2.

For problem A, while our model L3 solution in Listing 4.4 had a mean running time of 5.288 s, P31's noteworthy Copilot-unaided solution had a mean running time of 4.547 s beating our model solution by 15%. If we observe their solution in Listing 7.1, we note that their solution used the L3 optimization for problem A discussed in Chapter 4.1.4. Additionally, in lines 4 - 7 a map was used to associate each `fileId` with a vector of `fileCombos` for the associated file. The pre-processing in this step allowed them to sort each vector of `fileCombos` belonging to a file (line 9), open the file once (lines 11 - 12), process all the `fileCombos` (lines 13 - 16) and then close the file once (line 17). While the fundamental concept of the L3 optimization is present, some implementation details are slightly different and such may have contributed to the observed speed-up. It is also pertinent to mention that P31 had ideas to add another optimization that could have potentially reduced the running time of their code further if there had been sufficient time to debug their solution. From the comments from their code (removed for clarity) and the video analysis in Chapter 7.3.3, it would seem the potential improvement involved an application of `memcpy` [10] "to avoid overlaps".

For problem B, a noteworthy mention using the L1 solution with some model statement-level optimizations explained in Chapter 7.3 was P17's Copilot-unaided solution. Their code had a mean running time of 636.4 ms which was 9% slower when compared to the model L1 solution in Chapter 4.2.2 which had a mean running time of 581.4 ms. P17's code in Listing 7.2 is resemblant to the model L1 solution in Listing 4.6.

7.3 RQ2 - Do Copilot's suggestions sway developers to or from code with faster running time?

7.3.1 Approach

We wanted to understand how suggestions from Copilot swayed participants to produce code with slower or faster running times. To this end, we took the last snapshot of the participants' submitted code and categorized each participant's code for problems A and B. We labelled participants' code according to the optimizations discussed in Chapter 4. The author of this work and a collaborator separately looked through the source code for all participants and labelled each solution for problem A with either L0, L1, L2, or L3 to indicate the levels of optimizations that participants used. Similarly, for problem B,

```
bool compareByOffset(const FileCombo* a, const FileCombo* b) { return
 ↪ (a->offset < b->offset); }

void readFileCombos(std::vector<FileCombo> &fileCombos) {
  std::map<int, std::vector<FileCombo*>> combosByFile;
  for (FileCombo& combo : fileCombos) {
    combosByFile[combo.fileId].push_back(&combo);
  }
  for (auto combos : combosByFile) {
    std::sort(combos.second.begin(), combos.second.end(),
 ↪ compareByOffset);
    int previousOffset = 0-RECORD_SIZE-1;
    std::ifstream in;
    in.open(FILE_NAMES[combos.first]);
    for (FileCombo* combo : combos.second) {
      in.seekg(combo->offset);
      in.read(combo->buffer, RECORD_SIZE);
    }
    in.close();
  }
}
```

Listing 7.1: P31's L3 Solution to Problem A without Copilot

they were labelled as L0 or L1. Additionally, they also noted programming constructs that participants used that could potentially increase or decrease the running time and tried to group similar constructs. We term these "programming constructs" categories as statement-level optimizations. With the presence of these statement-level optimizations, we create another category to encompass the levels of optimizations described in Chapter 4 and call them concept-level optimizations from this point.

7.3.2 Statement Level Optimizations & Open-coding

After the author and the collaborator finished labelling participants' source files with the concept-level optimizations and statement-level optimizations, they came together to resolve disagreements on the concept-level optimizations and to discuss emerging patterns

```
1  void schedule() {
2    const int NUM_PER_THREAD = SIZE / THREAD_COUNT;
3    auto threadFunc = [ NUM_PER_THREAD](int index){
4      int start = NUM_PER_THREAD * index;
5      int end = min(NUM_PER_THREAD * (index+1), SIZE);
6      for (int j = start; j < end; j++) {
7        for (int k = 0; k < INIT_SRC_VAL; k++) {
8          src[j]--;
9          dst[j]++;
10       }
11     }
12   };
13   vector<thread> threads;
14   for (int i = 0; i < THREAD_COUNT; i++) {
15     threads.push_back(thread(threadFunc, i));
16   }
17   for (int i = 0; i < THREAD_COUNT; i++) {
18     threads[i].join();
19   }
20 }
```

Listing 7.2: P17's L1 Solution to Problem B without Copilot

in the statement-level optimizations and remarks. Upon resolving the disagreements they came up with a set of themes to encompass the statement-level optimizations. A summary of the categories of statement-level optimizations encoded for problem A and problem B are in Table 7.2 and Table 7.3 respectively.

Encoding	Summary
L*F	Used <fstream> [6] library for any of the concept-level optimizations L0, L1, L2, or L3
L*C	Used <cstdio> [3] library for any of the concept-level optimizations L0, L1, L2, or L3
L*U	Used <unistd.h> [22] and <fcntl.h> [5] libraries for any of the concept-level optimizations L0, L1, L2, or L3
NCLOSE	Did not close file
EXCEPT	Added `file.exceptions(...)` [18] to catch possible exceptions
ASSERT	Asserted that no error flags were set after file operations using `good()` [19] method and other assertions to ensure program correctness
READ_COMBO	Helper function for processing a single `fileCombo` in `fileCombos` and by calling `open()`, `seek()`, `read()`, and `close()` in order
BEGIN	Explicit seek from `std::ios_base::beg` [17] in call to `seekg()` [20]
OC_WITHIN	Opened and closed the files within the same loop as processing each `fileCombo` in `fileCombos`
BINARY	Added a "binary" flag to the `open` call using `std::ios::binary` [16] or similar
MAP	Used a `map` [21] to associate a file with all the `fileCombos` associated with that file

Table 7.2: Table of Statement-level Optimizations & remarks for Problem A

Encoding	Summary
NT	No threads used
ONET	Only one thread was used. Equivalent to not using threads
MISSING_LOOP	Failed to loop to decrement `src[i]` to zero and to increment `dst[i]` to `INIT_SRC_VAL`. This is an incorrect solution.
ITER_NAIVE	Made `SIZE X INIT_SRC_VAL` repeated calls to `dst[i].get()` or `src[i].get()` while decrementing `src[i]` and incrementing `dst[i]`
ITER_LESS_NAIVE	Made `SIZE` repeated calls to `src[i].get()` or `dst[i].get()` while decrementing `src[i]` and incrementing `dst[i]`
ITER_FAST	No calls to `src[i].get()` or `dst[i].get()` while decrementing `src[i]` and incrementing `dst[i]` but iterated up to `INIT_SRC_VAL`
2LOOPS	Decremented `src[i]` to 0 then incremented `dst[i]` to `INIT_SRC_VAL` instead of in lockstep
1LOOP	Decremented `src[i]` and incremented `dst[i]` in lockstep
SPLIT	`src[i]` is decremented using a separate thread and `dst[i]` is incremented using a separate thread
SPLIT2	Like SPLIT but `src[i]` decremented using 2 threads after being divided into 2 slices and `dst[i]` incremented using 2 threads after being divided into 2 slices
MANY_SPLIT	Spawned `SIZE` threads where each thread handled `src[i]` and `dst[i]`. There could be context switches since not enough threads on machine
LOCKS	Used locks.
RACET	Race conditions in thread spawning without locks leading to incorrect results
HARDT	Hardcoded thread spawning instead of dynamic based on `THREAD_COUNT`
PTHREAD	Used `pthread_create` and `pthread_join` [13] to create and join threads instead of `std::thread` methods
SPAWN_SEP	Spawned `THREAD_COUNT` threads to process `src[i]` then wait to finish then spawn another `THREAD_COUNT` threads for `dst[i]` then wait to finish
OPENMP	Used `parallel for` in OpenMP.

Table 7.3: Table of Statement-level Optimizations & remarks for Problem B

7.3.3 Video Analysis

Using the themes generated in Table 7.2 and Table 7.3, the author went through all 32 screen-shared recordings of participants solving the problem when Copilot was used and tracked the accepted suggestions or series of accepted suggestions that participants accepted that swayed them to the solutions that fit their themes.

7.3.4 Results

For problem A, where Copilot was used, 15 of the 16 correct solutions used the L0 naive implementation with the `<fstream>` [6] family of library functions and thus were categorized as L0F. Additionally, few remarks were made as most solutions only used the naive L0F implementation in Chapter 4.1.1. Some solutions were remarked as NCLOSE because they failed to close the files after reading from them. Some solutions also landed in the BINARY category. From the video analysis, it would seem that Copilot largely gave L0F suggestions, and participants simply accepted them without editing. Participants also seemed only to want to confirm that the sanity checks passed before declaring they were done with the problem.

In problem B, we notice a relatively balanced use of concept-level optimizations and varied use of statement-level optimizations and remarks for correct solutions with Copilot. From 14 of 16 source snippets with correct solutions, we note that 9 of those solutions used the L1 concept-level optimizations of avoiding false sharing. Notably, 1 of the 9 (P23) was classified as L1 because it avoided false sharing by using OpenMP to handle the multi-threaded execution. 2 of the 14 solutions were encoded as L0 even though false sharing was absent because they either did not use any threads (P7) or used only one thread (P3)

for the problem. 3 (P4, P11 and P19) of the 14 solutions were encoded as L0 because false sharing was present in their solutions. Additionally, statement-level remarks such as 2LOOPS or 1LOOP were prevalent in the solutions.

Moreover, ITER_NAIVE and ITER_FAST were also common categories that emerged. Rarer categories like OPENMP, ONET and NT also appeared in a few cases. From the video analysis, Copilot initially suggested incomplete snippets leaning toward L0. Participants would accept the snippets and try to get the rest of the solution to work by debugging. In other cases, participants wrote comments about dividing an array into `THREAD_COUNT` chunks, and Copilot would suggest snippets leaning towards L1.

7.3.5 Discussion

For problem A with Copilot, there was an interesting case where P22 was swayed via Copilot's suggestions to use L1U (Level 1 optimization but using the `<unistd.h>` [22] and `<fcntl.h>` [5] I/O functions). From the video analysis, we observe that the participant was largely responsible for coming up with concept-level L1 optimization in that they only declared a vector of file descriptors before the suggestions to use L1U with NCLOSE came along, which the participant accepted. However, P22 remarked that they "had to do more post-hoc checking" instead of "figuring out how to solve the problems"; that it was "a different approach of how they would solve the problem". We also note that while their solution used the L1 concept-level optimization, the average running time for their solution was **35.48 s** which was **15%** slower than our model L1 solution in Chapter 4.1.2. This running time difference may be related to differences in the I/O implementation details in the `<unistd.h>` and the `<fcntl.h>` libraries versus the `<fstream>` [6] library. See a snippet of P22's solution to problem A that was done with Copilot in Listing 7.3.

In solutions to problem B done with Copilot, we noticed that solution with the least mean running time at **677.8 ms** was from P12, which used the L1 concept-level optimization, and landed in the 1LOOP and ITER_LESS_NAIVE themes for the statement-level remarks. From the video analysis, the initial incomplete solutions accepted by the participant were leaning towards 1LOOP, NT and the incorrect solution of MISSING_LOOP. P12 was primarily responsible for implementing the code in the ITER_LESS_NAIVE statement-level remark because they "didn't think Copilot understood them[me] well when they[I] told it to increment or decrement" and "just gave up and wrote it themself[myself]". However, the L1 suggestion to split the thread into slices was accepted by the participant without much editing. P12 also remarked that "Copilot was useful", and they "usually just google" what Copilot would have suggested. We also note that their solution was 16% slower than

```
void readFileCombos(std::vector<FileCombo> &fileCombos) {
  std::vector<int> file_descriptors;
  for( auto fname : FILE_NAMES ) {
    int fd = open(fname.c_str(), O_RDONLY);
    file_descriptors.push_back( fd );
  }
  for( auto &combo : fileCombos ) {
    lseek(file_descriptors[combo.fileId], combo.offset, SEEK_SET);
    read(file_descriptors[combo.fileId], combo.buffer, RECORD_SIZE);
  }
}
```

Listing 7.3: P22's L1U Solution to Problem A with Copilot

our model L1 solution in Listing 4.6 which could be because the model L1 solution used ITER_FAST and 1LOOP statement level optimizations. See a snippet of P12's solutions to problem B that was done with Copilot in Listing 7.4.

```
int getChunkSize(int id) {
    return ( SIZE / THREAD_COUNT ) + 1;
}
int getChunkStart( int chunk ) {
    return chunk * getChunkSize(chunk);
}
int getChunkEnd( int chunk) {
    return min (getChunkStart(chunk) + getChunkSize(chunk), SIZE);
}

void* workerFunc(void *arg) {
  int id = (int64_t)arg;
  int start = getChunkStart(id);
  int end = getChunkEnd(id);

  for (int i = start; i < end; i++) {
    assert(src[i].get() == INIT_SRC_VAL);
    assert(dst[i].get() == 0);
    while ( src[i].get() != 0) {
```

```
1  void moveValues(int start, int end);
2
3  void moveValues(int start, int end){
4    int temp;
5    for(int i = start; i < end; i++){
6      temp = src[i].get();
7      for (int j = 0; j< temp; j++){
8        dst[i].operator++();
9        src[i].operator--();
10     }
11   }
12 }
13
14 void schedule() {
15   thread threads[THREAD_COUNT];
16   int thread_size = SIZE/THREAD_COUNT;
17   int start = 0;
18   int end = thread_size;
19   for(int i = 0; i < THREAD_COUNT; i++){
20     threads[i] = thread(moveValues, start, end);
21     start = end;
22     end += thread_size;
23   }
24   for(int i = 0; i < THREAD_COUNT; i++){
25     threads[i].join();
26   }
27 }
```

Listing 7.4: P12's L1 Solution to Problem B with Copilot

```
20       if ( INIT_SRC_VAL > 0 ) {
21         src[i]--;
22         dst[i]++;
23       } else {
24         src[i]++;
25         dst[i]--;
26       }
```

```
27      }
28      assert( src[i].get() == 0 );
29      assert( dst[i].get() == INIT_SRC_VAL );
30    }
31    return NULL;
32  }
33
34  void schedule() {
35    pthread_t threads[THREAD_COUNT];
36    int createdThreads = 0;
37    for( int i = 0; i < THREAD_COUNT; i++) {
38      int err = pthread_create(&threads[i], NULL, workerFunc,
    ↪   (void*)(int64_t) i);
39      if (err != 0) {
40        // ... COMMENTED OUT SO NOT AFFECT CSV GENERATION
41        break;
42      }
43      // ... COMMENTED OUT SO NOT AFFECT CSV GENERATION
44      createdThreads++;
45    }
46    for( int i = 0; i < createdThreads; i++) {
47      int err = pthread_join(threads[i], NULL);
48      if (err != 0) {
49        // ... COMMENTED OUT SO NOT AFFECT CSV GENERATION
50      }
51    }
52  }
```

Listing 7.5: P24's L1 Solution to Problem B with Copilot

Some interesting categories for statement level optimizations in problem B in Table 7.3 are worth taking a closer look at, notably, 2LOOPS, 1LOOP and ITER_NAIVE and ITER_FAST. Our model L1 solution in Listing 4.6 uses 1LOOP, ITER_FAST and also avoids false sharing and averages at a mean of **581.4 ms**. The closest Copilot-aided solution to the model solution in terms of running time was P12's in Listing 7.4 with a mean running time of **677.8 ms**. At a close second was P24 (See Listing 7.3.5) with a mean running time of **784.0 ms**, which avoided false sharing and used 1LOOP and ITER_NAIVE.

```cpp
void do_work(int begin, int end) {
  for (int i = begin; i < end; i++) {
    while (src[i].get() > 0) {
      src[i]--;
    }
    while (dst[i].get() < INIT_SRC_VAL) {
      dst[i]++;
    }
  }
}

void schedule() {
  std::thread threads[THREAD_COUNT];
  auto amount = SIZE / THREAD_COUNT;
  for (int i = 0; i < THREAD_COUNT; i++) {
    threads[i] = std::thread(do_work, i * amount, (i + 1) * amount);
  }
  for (int i = 0; i < THREAD_COUNT; i++) {
    threads[i].join();
  }
}
```

Listing 7.6: P27's L1 Solution to Problem B with Copilot

This difference in running time between the model solution in Listing 4.6 and P24's suggests that using ITER_FAST is better than using ITER_NAIVE to update the source and destination buffers when false sharing is avoided. If we also look at P27's Copilot-aided solution to problem B (See Listing 7.6), we notice that while it avoids false-sharing, it uses 2LOOPS and ITER_NAIVE which earns it a mean running time of **925.1 ms**. Comparing P24's with P27's solution suggests that using 2LOOPS instead of 1LOOP to update the source and destination buffers when false sharing is avoided could result in slower running time. On the other hand, if we look at solutions where false sharing was used, we note that both P11's (See Listing 7.7) and P19's (See Listing 7.8) Copilot-aided solutions had false sharing present. However, their solutions used 2LOOPS with ITER_NAIVE with a mean running time of **1434 ms** and 1LOOP with ITER_NAIVE with a mean running time of **6202 ms**, respectively. This difference in running time may suggest that using 1LOOP instead of 2LOOPS could result in slower running time when false sharing is present, which

is different from when false sharing is absent, as with P24's and P27's solutions.

```
1  void schedule() {
2    thread threads[THREAD_COUNT];
3    for (int i = 0; i < THREAD_COUNT; i++) {
4      auto lambda = [i]() {
5        for (int j = i; j < SIZE; j += THREAD_COUNT) {
6          while(src[j].get()!=0) {
7            src[j]--;
8          }
9          while (dst[j].get() != INIT_SRC_VAL) {
10           dst[j]++;
11         }
12       }
13     };
14     threads[i] = thread(lambda);
15   }
16   for (int i = 0; i < THREAD_COUNT; i++) {
17     threads[i].join();
18   }
19 };
```

Listing 7.7: P11's L0 Solution to Problem B with Copilot

```
1  void schedule() {
2    thread threads[THREAD_COUNT];
3    for (int i = 0; i < THREAD_COUNT; i++) {
4      threads[i] = thread([&](int i) {
5        for (int j = i; j < SIZE; j += 4) {
6          while (src[j].get() > 0) {
7            src[j]--;
8            dst[j]++;
9          }
10       }
11     }, i);
12   }
13   for (int i = 0; i < THREAD_COUNT; i++) {
14     threads[i].join();
15   }
16 }
```

Listing 7.8: P19's L0 Solution to Problem B with Copilot

Data from participants' solving		
runtime	Average of the running time of the 32 runs for each participant's code obtained in RQ1	positive floating-point
mode	Using Copilot (1) or not using Copilot (0)	1 or 0
time	How long participants took to solve the problem obtained from see 6.4 and 6.5	minutes
Data from the screening surveys		
devExp	How much programming experience do you have?	5-Point Likert-scale
familiarCPP	How familiar are you with the C++ programming language?	5-Point Likert-scale
Data from the first and second post-programming surveys		
understandProblem	How well did you understand the programming problem?	6-Point Likert-scale
confidentSolution	How confident are you in your solution to the programming problem?	6-Point Likert-scale
timeDebugging	How much time did you spend debugging your program (in minutes)?	number between 0 and 30
timeBrowser	How much time did you spend on your browser while solving the problem (in minutes)?	number between 0 and 30
Data from demographics questions included from the second post-programming survey		
familiarCopilot	How familiar are you with Github Copilot?	6-Point Likert-scale
familiarVSCode	How familiar are you with Visual Studio Code (VSCode)?	6-Point Likert-scale

Table 7.4: Table of Input Data to Correlation Matrix

	runtime	mode	time	understandProblem	confidentSolution	timeDebugging	timeBrowser	devExp	familiarCPP	familiarCopilot	familiarVSCode
runtime	1.00	0.46	-0.22	0.03	0.24	-0.15	-0.40	-0.39	-0.12	-0.01	0.21
mode	0.46	1.00	-0.58	-0.27	-0.22	-0.29	-0.72	-0.33	-0.48	0.04	0.01
time	-0.22	-0.58	1.00	-0.28	-0.26	0.67	0.75	0.17	0.33	-0.41	-0.33
understandProblem	0.03	-0.27	-0.28	1.00	0.59	-0.36	-0.06	-0.29	0.18	0.45	0.40
confidentSolution	0.24	-0.22	-0.26	0.59	1.00	-0.21	-0.09	-0.20	0.16	0.20	0.61
timeDebugging	-0.15	-0.29	0.67	-0.36	-0.21	1.00	0.52	0.18	0.19	-0.48	-0.28
timeBrowser	-0.40	-0.72	0.75	-0.06	-0.09	0.52	1.00	0.30	0.28	-0.09	-0.13
devExp	-0.39	-0.33	0.17	-0.29	-0.20	0.18	0.30	1.00	0.29	-0.23	-0.40
familiarCPP	-0.12	-0.48	0.33	0.18	0.16	0.19	0.28	0.29	1.00	0.08	-0.12
familiarCopilot	-0.01	0.04	-0.41	0.45	0.20	-0.48	-0.09	-0.23	0.08	1.00	0.13
familiarVSCode	0.21	0.01	-0.33	0.40	0.61	-0.28	-0.13	-0.40	-0.12	0.13	1.00

Table 7.5: Correlation matrix for problem A

7.4 RQ3 - Do characteristics of Copilot users influence the running time when it is used?

7.4.1 Approach

To give further context to RQ1, we wanted to see if certain characteristics of the participants influenced the running time with Copilot. To this end, we used data from the screening survey in Appendix B that determined participant eligibility, the post-programming survey for the first and second problem, demographics questions that were included as part of the second programming survey (see Appendix D), the running time data we obtained in Chapter 7.2, and the time participants spent on each problem obtained by the experimenter (see 6.4 and 6.5) and generated a Spearman correlation matrix to observe possible relationships. A summary of the data used in the correlation matrix is in Table 7.4. The data we used for this research question was only for valid solutions, i.e., solutions where the running time was obtained (See Chapter 7).

	runtime	mode	time	understandProblem	confidentSolution	timeDebugging	timeBrowser	devExp	familiarCPP	familiarCopilot	familiarVSCode
runtime	1.00	0.09	-0.21	-0.06	0.19	-0.04	-0.20	-0.30	-0.10	0.32	0.19
mode	0.09	1.00	-0.09	0.22	0.27	-0.34	-0.28	0.23	0.31	-0.04	0.05
time	-0.21	-0.09	1.00	-0.29	-0.21	0.62	0.67	-0.13	-0.08	-0.13	-0.19
understandProblem	-0.06	0.22	-0.29	1.00	0.42	-0.34	0.10	-0.06	-0.16	0.09	0.20
confidentSolution	0.19	0.27	-0.21	0.42	1.00	-0.26	-0.16	-0.02	-0.22	0.05	0.49
timeDebugging	-0.04	-0.34	0.62	-0.34	-0.26	1.00	0.49	-0.11	-0.13	-0.05	-0.10
timeBrowser	-0.20	-0.28	0.67	0.10	-0.16	0.49	1.00	-0.04	-0.26	-0.02	-0.21
devExp	-0.30	0.23	-0.13	-0.06	-0.02	-0.11	-0.04	1.00	0.29	-0.08	-0.31
familiarCPP	-0.10	0.31	-0.08	-0.16	-0.22	-0.13	-0.26	0.29	1.00	0.18	-0.06
familiarCopilot	0.32	-0.04	-0.13	0.09	0.05	-0.05	-0.02	-0.08	0.18	1.00	0.11
familiarVSCode	0.19	0.05	-0.19	0.20	0.49	-0.10	-0.21	-0.31	-0.06	0.11	1.00

Table 7.6: Correlation matrix for problem B

7.4.2 Results

In Table 7.5, we present a Spearman correlation matrix for problem A showing the coefficients. We observe a positive correlation between mode and running time of **0.46** for problem A. We also notice a negative correlation between developer experience and running time with a coefficient of **-0.39**. Also, we find that using Copilot is negatively correlated with the time spent on the browser while solving the problem with **-0.72** and time spent on the problem overall with **-0.58**. We also see that time spent on the browser was negatively correlated with the running time at **-0.40**.

Similarly, for problem B, we present a correlation matrix with the coefficients in Table 7.6. Like problem A, we also observe a positive correlation between using Copilot and running time with a coefficient of **0.09**. However, unlike problem A, this correlation is weaker. Developer experience is also negatively correlated with running time with a coefficient of **-0.30**. Also, like problem A, we find that mode is negatively correlated with time spent on the browser with a coefficient of **-0.28**, however, unlike problem A it has a weaker negative correlation with overall time spent on problem B with **-0.09**. We do see still see that the time spent on the browser was negatively correlated with running time at **-0.20**.

7.4.3 Discussion

While we notice some positive correlation between using Copilot and running time code for A, our results suggest that problem B may have a weaker correlation. This observation might suggest that using Copilot may have had less of an effect on running time for problem B than it did for problem A. To explain this, we look at the running times like we did in Chapter 7.2; however, in this case, we only consider the averages of all 32 iterations per participant, i.e., P22's snippet would be run 32 times, but we collect the average running time of the all 32 iterations as a single data point.

For problem A with Copilot, the "mean of means" (n1 = 16) running time for valid solutions was **34.86 s** and without Copilot (n2 = 14) was **26.02 s** (note that this is the same as what was reported in Chapter 7.2). When we compared the running time of Copilot (n1 = 16) versus not using Copilot (n2 = 14) via the same test in Chapter 7.2, we find the results still to be statistically significant ($p = 0.0172$).

For problem B with Copilot, the "mean of means" (n1 = 14) running time for valid solutions was **1898 ms** and without Copilot (n2 = 14) was **1628 ms** (also reported in Chapter 7.2). However, comparing running time for Copilot (n1 = 14) versus without Copilot (n2 = 14) was not statistically significant ($p = 0.667$). This could explain why there was such a strong positive correlation between mode and running time for problem A in Table 7.5 but a weak positive correlation for problem B in Table 7.6. This might also be due to the nature of the solutions to problem B involving a helper function and problem A not really needed any helper function unless a sort was involved. A more experienced developer or someone with more expertise with C++ may not need helper functions and simply use C++ lambdas [9] both for problem A (if using L2 or L3 concept-level optimization) and problem B.

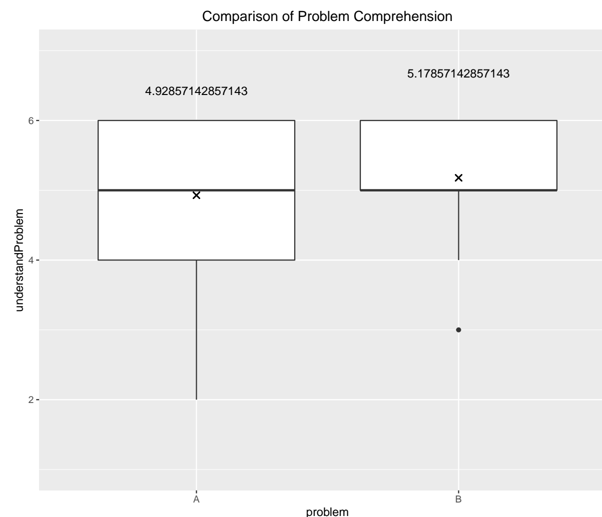

Figure 7.1: Plot of Problem Comprehension for Problem A vs B. "x" is the mean of the 6-Point Likert-Scale in Table 7.4 and Appendix D

We can also explain this by observing that using Copilot is negatively correlated with

developer experience and familiarity with C++ for problem A with coefficients of **-0.33** and **-0.48**. However, it is positively correlated for problem B with coefficients of **0.23** and **0.31**, respectively. This may suggest that developer experience and familiarity with C++ may contribute to reducing running time when developers do not use Copilot. The perceived difficulty of either problem may also explain this difference in problem A and problem B if we observe the correlation data of using Copilot with how well either problem was understood. In such a case, we notice a positive correlation with problem B at **0.22** and a negative correlation at **-0.27** with problem A. It may seem like Copilot may lull people into a false sense of security depending on the nature of the problem being solved because the running time had weak correlations with how well either problem was understood (**0.03** for problem A and **-0.06** for problem B). However, if we compare the problem comprehension regardless of Copilot usage, we observe a slightly different story (See Figure 7.1). We do not find a statistically significant difference ($p = 0.375$) in problem comprehension of problem A (n1 = 28) vs B (n2 = 28) when using the same non-parametric test in Section 7.2.

Chapter 8

Limitations and Takeaways

8.1 Limitations

We acknowledge the limitations of the representativeness of the programming problems that participants solved. While system developers may generally be aware of file system operations and multi-threading programming concepts, there could be more programming concepts that are not represented in our study that developers could have been more aware of. However, we argue that the domains of our problems are well represented in many undergraduate level Operating Systems courses.

Because we were looking at the running time of participants' code, one possible limitation could have been that participants did not feel like they had enough time to optimize their solution. However, on average all 32 participants spent approximately **17 minutes** of the 30 minutes allotted on problem A and **20 minutes** on problem B. The takeaway from this is that participants may have been satisfied with their solution at least 10 minutes before the time was up. Although it may seem that if participants were explicitly told to optimize their code, the results of our study would have been slightly different; however, we argue differently in that the crux of our research question is NOT how well Copilot can generate highly performant code compared to a human developer but if they are better off without it. The former research question would have required a participant pool of system developers who were performance experts and the experiment design and analysis would have to be different.

Additionally, we acknowledge that there could have been some participant selection bias

because in our recruitment stage we only selected participants that wanted to participate in the study. It could be that our results would have been slightly different if the developers that were unwilling to participate actually did the study. A workaround to recruit developers who would not have wanted to do the study because of negative perceptions about GitHub Copilot would be to omit the details about using Copilot until the session actually began. However, there are significant ethical concerns with this type of deception in controlled human studies so this workaround would likely not be implemented in the experiment design.

8.2 Takeaways

This work evaluated the performance of code generated by the self-proclaimed AI programming assistant GitHub Copilot by conducting a user study on systems programmers. While our study suggests that there may be some value in using Copilot to generate correct code, it also suggests that developers are better off without it when it comes to reducing running time. Even though we recommend that more experienced systems developers are better off without Copilot in their day-to-day tasks, we acknowledge that Copilot and others like it are a step in the right direction for programmers. With GitHub Copilot being one of the first production-ready programming assistants that are gaining ubiquity in modern software development, more research needs to be done on not only acknowledging its limitations but also its strengths. As our study is one of the first to evaluate Copilot from a performance perspective in systems programming, we hope future work can build off our experiment design to more granularly analyze Copilot's suggestions when developers use it.

References

[1] About github copilot. https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot. Accessed: 2022-11-08.

[2] Common systems programming optimizations & tricks. https://paulcavallaro.com/blog/common-systems-programming-optimizations-tricks/. Accessed: 2022-10-04.

[3] cstdio - c library to perform input/output operations. https://cplusplus.com/reference/cstdio/. Accessed: 2022-11-14.

[4] Discord. https://discord.com/. Accessed: 2022-10-02.

[5] fcntl.h - file control options. https://pubs.opengroup.org/onlinepubs/7908799/xsh/fcntl.h.html. Accessed: 2022-11-14.

[6] fstream - input/output file stream class. https://cplusplus.com/reference/fstream/fstream/. Accessed: 2022-11-14.

[7] gcc-9 9.3.0-17ubuntu1 20.04 source package in ubuntu. https://launchpad.net/ubuntu/+source/gcc-9/9.3.0-17ubuntu1~20.04. Accessed: 2022-11-12.

[8] Internet relay chat (irc). https://www.irchelp.org/. Accessed: 2022-10-02.

[9] Lambda expressions. https://en.cppreference.com/w/cpp/language/lambda. Accessed: 2022-11-17.

[10] memcpy - copy block of memory. https://cplusplus.com/reference/cstring/memcpy/. Accessed: 2022-10-27.

[11] Modaledit - modal editing in vs code. https://marketplace.visualstudio.com/items?itemName=johtela.vscode-modaledit. Accessed: 2022-10-01.

[12] Operating systems development. https://wiki.osdev.org/Expanded_Main_Page. Accessed: 2022-10-03.

[13] pthreads(7) — linux manual page. https://man7.org/linux/man-pages/man7/pthreads.7.html. Accessed: 2022-11-14.

[14] Serenity os. https://serenityos.org/. Accessed: 2022-10-03.

[15] Skift os. https://skiftos.org/. Accessed: 2022-10-03.

[16] std::ios_base::openmode. https://en.cppreference.com/w/cpp/io/ios_base/openmode. Accessed: 2022-11-14.

[17] std::ios_base::seekdir. https://en.cppreference.com/w/cpp/io/ios_base/seekdir. Accessed: 2022-11-14.

[18] std::ios::exceptions. https://cplusplus.com/reference/ios/ios/exceptions/. Accessed: 2022-11-14.

[19] std::ios::good. https://cplusplus.com/reference/ios/ios/good/. Accessed: 2022-11-14.

[20] std::istream::seekg. https://cplusplus.com/reference/istream/istream/seekg/. Accessed: 2022-11-12.

[21] std::map. https://cplusplus.com/reference/map/map/. Accessed: 2022-11-14.

[22] unistd.h - standard symbolic constants and types. https://pubs.opengroup.org/onlinepubs/7908799/xsh/unistd.h.html. Accessed: 2022-11-14.

[23] Visual studio code remote - ssh. https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-ssh. Accessed: 2022-10-02.

[24] Visual studio code wsl. https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-wsl. Accessed: 2022-10-02.

[25] Vscodevim - vim emulation for visual studio code. https://marketplace.visualstudio.com/items?itemName=vscodevim.vim. Accessed: 2022-10-01.

[26] Wilcoxon rank sum and signed rank tests. https://stat.ethz.ch/R-manual/R-devel/library/stats/html/wilcox.test.html. Accessed: 2022-10-16.

[27] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[28] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

[29] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 213–226, USA, 2008. USENIX Association.

[30] Sheheeda Manakkadu and Sourav Dutta. Bandwidth based performance optimization of multi-threaded applications. In *2014 Sixth International Symposium on Parallel Architectures, Algorithms and Programming*, pages 118–122, 2014.

[31] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. An empirical cybersecurity evaluation of github copilot's code contributions. *CoRR*, abs/2108.09293, 2021.

[32] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Security implications of large language model code assistants: A user study, 2022.

[33] Yongseok Son, Heon Young Yeom, and Hyuck Han. Optimizing i/o operations in file systems for fast storage devices. *IEEE Transactions on Computers*, 66(6):1071–1084, 2017.

[34] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery.

[35] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *ACM Trans. Softw. Eng. Methodol.*, 31(2), mar 2022.

APPENDICES

Appendix A

Full Description of the Problems Given to Participants

A.1 Problem A

```
1  /**
2   * PLEASE READ ENTIRE FILE CAREFULLY BEFORE YOU BEGIN.
3   * Your goal in this exercise is to read
4   * many records from large text files and
5   * write them to a buffer. A record
6   * is a series of bytes in a large text file.
7   *
8   */
9
10 #include <algorithm>
11 #include <fstream>
12 #include <iostream>
13 #include <random>
14 #include <string>
15 #include <vector>
16
17 using namespace std;
18
19 /** The size of each record in bytes*/
20 #define RECORD_SIZE 5000
```

```
21
22  /** The number of records to read*/
23  #define NUM_RECORDS 500000
24
25  /** Global vector of strings which are the file names of some large text
    ↪  files
26   * in your working directory. You may assume that each file is at least
    ↪  1GB (1e9
27   * bytes) in size and contains random text.
28   * Because it is impractical to send several 1GB files, we have provided
    ↪  10MB
29   * (1e7 bytes) files instead for easier debugging
30   * DO NOT EDIT!
31   */
32  const std::vector<std::string> FILE_NAMES = {
33      "large_file_1.txt", "large_file_2.txt", "large_file_3.txt"};
34
35  /** The size of the file in bytes. Ideally this should be
36   * 1e9 bytes (1GB) but it is impractical to send out
37   * such a large file.
38   * DO NOT EDIT!
39   */
40  #define FILE_SIZE 1e7
41
42  /** A struct representing a file combination
43   * DO NOT EDIT!
44   */
45  struct FileCombo {
46    int fileId; /** A valid index in `FILE_NAMES` */
47    int offset; /** A valid seek position for a file at
    ↪  `FILE_NAMES[fileId]`*/
48    char buffer[RECORD_SIZE + 1]; /** The output buffer */
49  };
50
51  std::vector<FileCombo> createCombos();
52  void sanityCheck(std::vector<FileCombo> &combos);
53
54  void readFileCombos(std::vector<FileCombo> &fileCombos);
```

```
55
56 /**
57  * Implement the `readFileCombos` function that reads all the records from
   ↪  the
58  * vector of file combinations `fileCombos` and writes each record to the
59  * `buffer` member of the `FileCombo` struct. Each record is `RECORD_SIZE`
60  * bytes. You may assume that reading `RECORD_SIZE` bytes from any file in
61  * `FILE_NAMES` starting at any seek position, `offset` in `fileCombos`
   ↪  will
62  * never throw an error.
63  *
64  */
65
66 /**
67  * Example
68  *
69  * * * * * * example_file.txt * * * * * *
70  * fourLettTextUsedHereAlsoHere
71  * * * * * * EOF * * * * * *
72  *
73  * * * * * * program.cpp * * * * *
74  * #define RECORD_SIZE 4
75  * #define NUM_RECORDS 4
76  * const std::vector<std::string> FILE_NAMES = {"example_file.txt"};
77  *
78  * void readFileCombos(std::vector<std::string> &fileCombos,
79  *                     std::vector<char[RECORD_SIZE + 1]> &buffer);
80  *
81  * int main() {
82  *  std::vector<FileCombo> fileCombos = {
83  *    {.fileId = 0, .offset = 4,  .buffer = ""},
84  *    {.fileId = 0, .offset = 20, .buffer = ""},
85  *    {.fileId = 0, .offset = 8,  .buffer = ""},
86  *    {.fileId = 0, .offset = 24, .buffer = ""},
87  *  };
88  *  readFileCombos(&fileCombos);
89  *  return 0;
90  * }
```

```
91   * * * * * * EOF * * * * * *
92   *
93   * fileCombos is now
94   *  {
95   *    {.fileId = 0, .offset = 4,  .buffer = "Lett"},
96   *    {.fileId = 0, .offset = 20, .buffer = "Also"},
97   *    {.fileId = 0, .offset = 8,  .buffer = "Text"},
98   *    {.fileId = 0, .offset = 24, .buffer = "Here"},
99   *  }
100  *
101  */
102
103 void readFileCombos(std::vector<FileCombo> &fileCombos) {
104   // YOUR CODE GOES HERE
105 }
106
107 int main() {
108   // Main function
109   std::vector<FileCombo> fileCombos = createCombos();
110
111   readFileCombos(fileCombos);
112
113   // Uncomment the below line to test for correctness
114   sanityCheck(fileCombos);
115   return 0;
116 }
117 /**
118  * Generates vector of `FileCombo` structs of size `NUM_RECORDS`
119  * where each `fileId` member is a random index from `FILE_NAMES`
120  * and each `offset` member is a random valid seek position for the file
    ↪  at
121  * `FILE_NAMES[fileId]`.
122  * DO NOT EDIT!
123  */
124 std::vector<FileCombo> createCombos() {
125   std::random_device rd;
126   std::mt19937 gen(rd());
127   std::uniform_int_distribution<int> fileIdDis(0, FILE_NAMES.size() - 1);
```

```
128   std::uniform_int_distribution<int> offsetDis(0,
129                                                   FILE_SIZE - (2 *
    ↪  RECORD_SIZE));
130
131   std::vector<FileCombo> combos;
132   for (int i = 0; i < NUM_RECORDS; ++i) {
133     FileCombo fc = {.fileId = fileIdDis(gen), .offset = offsetDis(gen),
    ↪  ""};
134     combos.push_back(fc);
135   }
136   return combos;
137 }
138
139 /**
140  * Checks that the `buffer` member of each `FileCombo` struct in `combos`
141  * is the correct record.
142  * DO NOT EDIT!
143  */
144 void sanityCheck(std::vector<FileCombo> &combos) {
145   printf("Running Sanity Checks...\n");
146   for (int i = 0; i < NUM_RECORDS; ++i) {
147     std::ifstream in;
148     in.open(FILE_NAMES[combos[i].fileId]);
149     in.seekg(combos[i].offset);
150     char temp[RECORD_SIZE + 1] = "";
151     in.read(temp, RECORD_SIZE);
152     in.close();
153     if (std::string(combos[i].buffer) != std::string(temp)) {
154       printf("combos[%d].fileId = %d\n", i, combos[i].fileId);
155       printf("combos[%d].offset = %d\n", i, combos[i].offset);
156       printf("combos[%d].buffer = %s\n", i, combos[i].buffer);
157       printf("instead of\ncombos[%d].buffer = %s\n", i, temp);
158       printf("Check Failed!\n");
159       return;
160     }
161   }
162   printf("All Checks passed!\n");
163 }
```

Listing A.1: Full CPP File for Problem A

A.2 Problem B

```
/**
 * PLEASE READ ENTIRE FILE CAREFULLY BEFORE YOU BEGIN.
 * Your goal in this exercise is to apply multithreaded programming to
 * set all the values in a source array buffer to zero while setting all
↪  the
 * values in a destination array buffer to a particular value. However,
↪  you are
 * only allowed to increment or decrement the values in the respective
↪  array
 * buffers. Assignment operations (move and copy) are not allowed on
↪  either the
 * source array buffer or the destination array buffer.
 *
 */

#include <algorithm>
#include <iostream>
#include <thread>

using namespace std;

/**
 * The intial value of every `val` member in
 * `src` array for the problem statement
 * 2 ^ 17 = 131072
 */
const int INIT_SRC_VAL = (1 << 17);

/**
 * The size of the arrays in the problem statement
 * 2 ^ 11 = 2048
 */
```

```
const int SIZE = (1 << 11);

/**
 * Number of threads you are permitted to use
 * for the problem statement
 */
const int THREAD_COUNT = 4;

/**
 * A struct containing an integer that can only
 * be incremented or decremented.
 * DO NOT EDIT!
 */
struct Item {
 private:
  /** Private integer member*/
  int val;

  /** Private Copy and Move Cosntructors
   * to prevent move and copy operations
   */
  Item(const Item&);
  Item(Item&&);
  Item& operator=(const Item&);
  Item& operator=(Item&&);

 public:
  Item() { val = 0; }
  Item(int i) { val = i; }
  /** Returns the integer member `val`*/
  int get() { return val; }

  /** Increments the integer member `val`*/
  void operator++() { ++val; }
  void operator++(int) { val++; }

  /** Decrements the integer member `val`*/
  void operator--() { --val; }
```

```
67    void operator--(int) { val--; }
68  };
69
70  /**
71   * Global array of `Item` structs where each
72   * `val` member is initialized to `INIT_SRC_VAL`
73   */
74  Item src[SIZE];
75  void initSrc();
76
77  /**
78   * Global array of `Item` structs where each
79   * `val` member is initialized to 0
80   */
81  Item dst[SIZE];
82
83  void sanityCheck();
84
85  void schedule();
86  /**
87   * Using `THREAD_COUNT` threads, implement the function `schedule`
88   * which mutates the `val` member for each `Item` in `src` to 0 and the
89   * `val` member for each `Item` in `dst` to `INIT_SRC_VAL`. The `Item`
90   * struct only supports a `get()` method and increment and decrement
    ↪  operations
91   * and a few initialization constuctors. See the `Item` struct for more
    ↪  details.
92   *
93   * `src[0]++`, `++src[0]`, `src[0]--` or `--src[0]` are valid.
94   * `dst[i]++`, `++dst[i]` `dst[0]--` or `--dst[0]` are also valid.
95   * `Item p = Item(54)` is valid.
96   * `Item items[4] = {Item(1), Item(2), Item(3), Item(4)}` is valid.
97   * `items[4] = items[3]` is not valid.
98   * `items[4] = Item(32)` is not valid.
99   * `src[0] = 0` or `dst[0] = INIT_SRC_VAL` or `Item temp = src[0]` or
    ↪  similar
100  * are not valid.
101  *
```

```
102  */
103
104 /**
105  * Example
106  * 2 ^ 10 = 1024
107  * INIT_SRC_VAL = (1 << 10)
108  *
109  * 2 ^ 2 = 4
110  * SIZE = (1 << 2)
111  *
112  *
113  * Initial values for `src` and `dst`
114  * src = {
115  *  {.val = 1024},
116  *  {.val = 1024},
117  *  {.val = 1024},
118  *  {.val = 1024}
119  * }
120  *
121  * dst = {
122  *  {.val = 0},
123  *  {.val = 0},
124  *  {.val = 0},
125  *  {.val = 0}
126  * }
127  *
128  * Make the call to `schedule`
129  * schedule()
130  *
131  * Values for `src` and `dst` after `schedule` is called
132  * src = {
133  *  {.val = 0},
134  *  {.val = 0},
135  *  {.val = 0},
136  *  {.val = 0}
137  * }
138  *
139  * dst = {
```

```
140  *  {.val = 1024},
141  *  {.val = 1024},
142  *  {.val = 1024},
143  *  {.val = 1024}
144  * }
145  *
146  */
147
148 void schedule() {
149   // YOUR CODE GOES HERE
150 }
151
152 int main() {
153   // Main function
154   initSrc();  // DO NOT USE
155
156   schedule();
157
158   // Uncomment the line below to test for correctness
159   sanityCheck();
160   return 0;
161 }
162
163 /**
164  * Initializes the `val` member for each `Item` in `src`
165  * to `INIT_SRC_VAL`. Called in `main` once and never again.
166  * DO NOT EDIT or CALL!
167  */
168 void initSrc() {
169   for (int i = 0; i < SIZE; ++i) {
170     for (int j = 0; j < INIT_SRC_VAL; ++j) {
171       ++src[i];
172     }
173   }
174 }
175 /**
176  * Verifies that the `val` member for each `Item` in `src`
177  * is 0 and the `val` member for each `Item` in `dst`
```

```
178    * is `INIT_SRC_VAL`.
179    */
180   void sanityCheck() {
181     printf("Running Sanity Checks...\n");
182     bool isFail = false;
183     for (int i = 0; i < SIZE; ++i) {
184       if (src[i].get() != 0) {
185         printf("src[%d].get() = (%d) instead of (%d)\n", i, src[i].get(),
      ↪  0);
186         isFail = true;
187       }
188       if (dst[i].get() != INIT_SRC_VAL) {
189         printf("dst[%d].get() = (%d) instead of (%d)\n", i, dst[i].get(),
190                INIT_SRC_VAL);
191         isFail = true;
192       }
193
194       if (isFail) {
195         printf("Check Failed!\n");
196         return;
197       }
198     }
199
200     printf("All Checks Passed!\n");
201   }
```

Listing A.2: Full CPP File for Problem B

Appendix B

Screening Survey Given to Participants to Determine Eligibility

How much programming experience do you have?	
No experience	1
Less than 1 year	2
Between 1 year and 5 years (excluding)	3
Between 5 years and 10 years (excluding)	4
10 years or more	5
Do you have access to GitHub Copilot on VSCode?	
No	1
No, but I have applied	2
Yes	3
Are you employed by Open AI or Github, or were you involved with the development of Github Copilot?	
No	1
Yes, please explain:	TEXT_FIELD
Have you taken a systems course like Operating Systems, Distributed Systems, or Computer Networks?	
No	1
Yes, please enter the course title:	TEXT_FIELD
How familiar are you with the C++ programming language?	
Not familiar at all	1
Slightly familiar	2
Moderately familiar	3
Very familiar	4
Extremely familiar	5

Table B.1: Screening Survey

Appendix C

Tutorial

This chapter goes into the details of the tutorial that the experimenter gave the participants before they were given the first programming problem.

The facilitator started sharing their screen and got confirmation that the participant could see the screen. They then demonstrated unzipping a ZIP file wherein a CPP was compressed and opening the uncompressed folder in VSCode. Following that, they checked that all VSCode extensions were disabled except for the Copilot extension for VSCode. The experimenter then opened their browser and showed that they could easily switch between their browser and VSCode. They also mentioned that participants are allowed to use their browser when solving either programming problem so there was no ambiguity as to whether they were allowed to use their browser or not. The experimenter activated Copilot and demonstrated using Copilot suggestions for the prompt "print hello world 8 times but on even times append an emoji". The experimenter demonstrated accepting a suggesting, rejecting a suggestion, toggling between the next and previous suggestions, and then viewing all suggestions using the "Open GitHub Copilot" menu option. See Figure 2.1 for the prompt and Copilot's options.

They then demonstrated compiling the program with `g++ hello.cpp -o hello -std=c++17` and running the program. The facilitator stressed that `-std=c++17` must be used and that if participants were on a Linux machine or were using Windows Subsystem for Linux (WSL), they could add the `-lpthread` flag like so `g++ hello.cpp -o hello -std=c++17 -lpthread`. After compiling and running the program, the experimenter demonstrated zipping up the entire folder and sending the ZIP file via the online conferencing platform. The experimenter then demonstrated deactivating Copilot, closing VSCode and closing the browser used for the current problem. It was emphasized that this "cleanup" must be done after each problem

is completed.

Appendix D

Programming Surveys Given to Participants After Solving a Problem

D.1 First Programming Survey

Have you seen this programming problem before?	
Prefer not to answer No Maybe Yes	1 2 3 4
Have you solved this programming problem before?	
Prefer not to answer No Maybe Yes	1 2 3 4
How well did you understand the programming problem?	
Prefer not to answer Not well at all Slightly well Moderately well Very well Extremely well	1 2 3 4 5 6
How confident are you in your solution to the programming problem?	
Prefer not to answer Not confident at all Slightly confident Moderately confident Very confident Extremely confident	1 2 3 4 5 6
How much time did you spend debugging your program (in minutes)?	
TEXT_FIELD	
How much time did you spend on your browser while solving the problem (in minutes)?	
TEXT_FIELD	
What resources did you use to help you solve the problem (i.e. StackOverflow, GeeksForGeeks etc.)?	
TEXT_FIELD	

Table D.1: First Programming Survey

D.2 Second Programming Survey

Have you solved this programming problem before?	
Prefer not to answer No Maybe Yes	1 2 3 4

Have you solved this programming problem before?	
Prefer not to answer	1
No	2
Maybe	3
Yes	4
How well did you understand the programming problem?	
Prefer not to answer	1
Not well at all	2
Slightly well	3
Moderately well	4
Very well	5
Extremely well	6
How confident are you in your solution to the programming problem?	
Prefer not to answer	1
Not confident at all	2
Slightly confident	3
Moderately confident	4
Very confident	5
Extremely confident	6
How much time did you spend debugging your program (in minutes)?	
TEXT_FIELD	
How much time did you spend on your browser while solving the problem (in minutes)?	
TEXT_FIELD	
What resources did you use to help you solve the problem (i.e. StackOverflow, GeeksForGeeks etc.)?	
TEXT_FIELD	
What is your highest level of education at the moment (degree awarded/ongoing)?	
Prefer not to answer	1
Less than high school	2
High school graduate	3
Some college	4
2 year degree	5
4 year degree	6
Professional degree (Masters)	7
Doctorate	8
Other, please specify:	TEXT_FIELD
What is your current employment?	
TEXT_FIELD	

How familiar are you with Github Copilot?	
Prefer not to answer	1
Not familiar at all	2
Slightly familiar	3
Moderately familiar	4
Very familiar	5
Extremely familiar	6
How familiar are you with Visual Studio Code (VSCode)?	
Prefer not to answer	1
Not familiar at all	2
Slightly familiar	3
Moderately familiar	4
Very familiar	5
Extremely familiar	6
Have you taken a security course like Computer Security & Privacy, Cryptography, or Network Security?	
No	1
Yes, please enter the course title:	TEXT_FIELD
How familiar are you with the Python programming language?	
Prefer not to answer	1
Not familiar at all	2
Slightly familiar	3
Moderately familiar	4
Very familiar	5
Extremely familiar	6
How familiar are you with the Java programming language?	
Prefer not to answer	1
Not familiar at all	2
Slightly familiar	3
Moderately familiar	4
Very familiar	5
Extremely familiar	6

Table D.2: Second Programming Survey Plus Demographics