The Value of Slicing while Debugging

Margaret Ann Francel
Department of Mathematics and Computer Science
The Citadel
Charleston, South Carolina USA 29409
francelm@citadel.edu

Spencer Rugaber
College of Computing
Georgia Institute of Technology
Atlanta, Georgia USA 30332
spencer@cc.gatech.edu

Abstract

The paper describes a study that explored the relationship of program slicing to (1) code understanding gained while debugging, and to (2) a debugger's ability to localize the program fault area. The study included two experiments. The first experiment compared the program understanding abilities of two classes of debuggers: those who slice while debugging and those who do not. For debugging purposes, a slice can be thought of as a minimal subprogram of the original code that contains the program faults. Those who only examine statements within a slice for correctness are considered slicers; all others are considered non-slicers. Using accuracy of subprogram construction as a measure of understanding, it was determined that slicers have a better understanding of the code after debugging. The second experiment compared debugger fault localization abilities before and after a training session on how to use slicing in debugging. Using time as a measure of ability, it was shown that slicing while debugging improves a debugger's ability to localize the program fault area.
Keywords: program slicing, reverse engineering, debugging

1 INTRODUCTION

Debugging code comprises a significant portion of the software development and maintenance process. Yet no "best" method for debugging programs is known. However, the two experiments described in this paper support the premise that slicing while debugging has a positive effect on the debugging process. The first experiment described indicates that debuggers who use slicing while debugging have a better understanding of the code at the end of the debugging process than those who do not use slicing. The second experiment described indicates that debuggers are better able to localize the program fault area if they use slicing while debugging, then if they debug without slicing.

Debugging is the task of identifying faults in code. A goal of the *debugger*, the person who debugs the code, is to localize the fault area of the code, and at the same time, to develop an understanding of the program so that an adequate correction can be made. Working toward this goal is a labor-intense and time-consuming activity. As a result, it is critical to identify debugging strategies that lead to quick reduction of the code fault area coupled with increased understanding.

Above, and throughout the paper, we use the term *fault* as defined in the *IEEE Standard Glossary of Software Engineering Terminology* [2]. The glossary distinguishes between a program error, fault and failure. A mistake in the human thought process made during the construction of a program is called an *error*. Evidence of errors comes through program *failures*, typically incorrect output values, unexpected program termination, or nonterminating execution. It is often the case that the root cause of a failure can be traced to a small area of a program. If so, that area is said to contain a *fault*. It is important to note that sometimes program failures are indications of global problems such as mistaken assumptions or inappropriate architectural decisions. In such cases, it is misleading to assume that editing a small area of the program will prove sufficient to correct an error.

```
( 1)   // This program processes a list of integers input interactively during the
( 2)   // program run. The program counts the number of non-negative integers in the
( 3)   // list, and finds the maximum and minimum negative integer in the list.

( 4)   #include<iostream.h>

( 5)   int main ()
( 6)   {
( 7)       int current_value;
( 8)       int current_maximum;
( 9)       int current_minimum;
(10)       int non_negative_count;
(11)       int lcv;
(12)       int size;
(13)       int integer_array[100];

(14)       size = 0;
(15)       cout << "Input an integer or –999999 to stop\n";
(16)       cin >> current_value;
(17)       while (current_value != -999999)
(18)       {
(19)           integer_array[size] = current_value;
(20)           size++;
(21)           cout << "Input an integer or –999999 to stop\n";
(22)           cin >> current_value;
(23)       }

(24)       current_minimum = 0;
(25)       current_maximum = 0;
(26)       non_negative_count = 0;
(27)       for (lcv = 0; lcv < size; lcv++)
(28)       {
(29)           if (integer_array[lcv] < 0)
(30)           {
(31)               if (integer_array[lcv] < current_minimum)
(32)                   current_minimum = integer_array[lcv];
(33)               if (integer_array[lcv] > current_maximum)
(34)                   current_maximum = integer_array[lcv];
(35)           }
(36)           else
(37)               non_negative_count++;
(38)       }

(39)       cout << endl <<endl <<endl;
(40)       cout << "The number of non-negative numbers in the list was "
                << non_negative_count
                << endl;
(41)       cout << "The maximum negative number in the list was "
                << current_maximum
                << endl;
(42)       cout << "The minimum negative number in the list was "
                << current_minimum
                << endl;
(43)   }
```

Figure 1 : A Buggy C++ Program (Fault Area: Line 25)

At the start of debugging, as far as the debugger is concerned, the code fault area can be anywhere in a program. It is the task of the debugger to reduce this range as much as possible. No one method of code reduction is favored universally by people who debug programs. Rather different people prefer different methods. However, a large number of experts use a code reduction method called program slicing while debugging [8].

Mark Weiser developed the concept of slicing in the early 1980's [7]. Slicing is based on the flow of data through a program. Formally, a *slice* of code with respect to a statement *S* and variable *v* consists of exactly those statements of the code that might affect the value of *v* at statement *S*. Table 1 shows three slices taken from the buggy C++ program shown in Figure 1.

If the variable *v* in the output statement *S* contains an incorrect value, then the slice on *v* at *S* will contain the program fault causing the incorrect output value. This is true because program failures are

indications of program faults and a slice on variable *v* with respect to statement *S* contains all statements that might affect the value of *v* at *S*. This makes slices based on statements that are outputting incorrect values especially helpful in debugging. For example in the buggy program of Figure 1, once one determines that exactly one output value is incorrect and that that value is being output in statement #41, the code fault area of the program can be reduced to the slice on *current_maximum* at statement #41 (slice 2 of Table 1). This decreases the size of the fault area from the original by ten statements or 25%.

Table 1: Three Static Slices from the Buggy Code of Figure 1

Statement	Variable	Statements, by number, in slice
40	non_negative_count	4,5,6,7,10,11,12,13,14,15,16,17, 18,19,20,21,22,23,26,27,28,29, 36,37,38,39,40,43
41	current_maximum	4,5,6,7,8,11,12,13,14,15,16,17, 18,19,20,21,22,23,25,27,28, 29,30,33,34,35,38,39,41,43
42	current_minimum	4,5,6,7,9,11,12,13,14,15,16,17, 18,19,20,21,22,23,24,27,28,29, 30,31,32,35,38,39,42,43

Knowing which output statements produce correct values can also help in fault area reduction. A program *dice*, first defined by Weiser and Lyle [9], is a slice on one set of program variables at a statement minus a slice on a second set of variables at the same or another statement. If one is willing to assume correct output implies correct code[1], slicing a program on incorrect output variables and then dicing on correct output variables can reduce the program fault area even more than slicing on incorrect output variables alone does. In the above example, slicing on *current_maximum* and then dicing on {*current_min*, *positive_count*} gives a program fault area of five statements, {8,25,33,34,41}. This is a decrease in size of 87.5% from the original fault area.

Slicing and dicing are code reduction methods. But do they lend themselves to increased understanding of the program code and to improved fault localization? The purpose of the exploratory study described in this paper is to show that (1) a strong relationship exists between debugging with slicing and program understanding, and (2) using slicing while debugging improves a debugger's ability to localize the program fault area.

Since the concept of slicing was first formalized in the early 1980's, it has been the focus of much research. However, the emphasis of this research has been tool building and the issues and problems related to this activity. Applications of slicing and the advantages of using slicing have been largely neglected. This becomes unmistakably evident when one reviews any of the reference lists on slicing. (Note: Several of these lists can be accessed through the Algorithmic and Automatic Debugging Home Page [1].) For example, a total of only 3 of the 111 articles appearing in the bibliography of slicing maintained by Krinke [4] are concerned with issues of slicing applications or the advantages of using slicing. The same lack is found in the reference list of slicing articles maintained by Lyle [5]. Again, here we find the three references from Krinke's list that address issues of slicing applications or the advantages of using slicing but no others. One of these three references is to Weiser's dissertation [6]. The second documents, with empirical data, that experts naturally slice when debugging [8]. The third shows, using empirical data, that debuggers who were given the original program with the faulty dice highlighted took less time to localize the program fault area than debuggers who are given the original program with no highlighting [9]. Clearly issues related to applications of slicing and the advantages of using slicing are an important piece in the overall slicing picture. Despite this, as indicated above, the amount of research devoted to these areas is small. Further investigation in these areas is needed to lend a more balanced understanding to the overall picture.

Below we explore (1) the relationship between slicing while debugging and code understanding, and

[1] There are exceptions to this assumption. For example, if a statement contains the expression a+b where it should contain a*b, no program failure occurs if the code is tested with the value 2 assigned to both a and b. However, it is hoped that debuggers base their conclusions that a statement produces correct output on suites of tests rather than on single tests, which makes the above type exception less likely to happen.

(2) the effect of slicing while debugging on fault localization. We find that (1) debuggers who use slicing have a better understanding of the program code after debugging than those who did not use slicing, and (2) using slicing while debugging improves a debugger's ability to localize the program fault area.

2 EXPERIMENT I

Experiment I was conducted to test the hypothesis that slicing while debugging increases program understanding. For the experiment, each subject was first asked to debug a program, recording as they went the program statements they examined for correctness. As a follow-up activity, subjects were asked to construct from the program code the minimal subprogram that produced the incorrect output. Besides the statement list generated during debugging and the subprogram constructed in the follow-up task, start and stop times were recorded for both the debugging and subprogram construction activities of the experiment. The experiment data was used to investigate the hypothesis that program debuggers who slice while debugging code gain better understanding of the program code than those program debuggers who do not slice during debugging.

2.1 Subjects

The subjects were seventeen senior computer science majors at a small liberal arts college in the Southeastern United States. Each volunteered for the experiment. At the time of the experiment, all subjects had completed at least three computer science courses whose major emphasis was programming. All had coded using Pascal in these courses. Also at the time of the experiment, each subject had completed between three and seven other computer science courses. None of the subjects had been exposed to the concepts of slicing or dicing. All subjects received extra credit in a senior seminar course for their participation in the experiment.

2.2 Procedure

Each subject participated in a single one-on-one session with the experimenter. There were no time limits imposed on any of the experiment activities, so sessions ran anywhere from forty minutes to two and a half hours. The experimenter was available to the subject throughout the entire session. Each session consisted of three phases:

Phase I:	Instructions and practice
Phase II:	Program debugging
Phase III:	Subprogram construction

Phase I: Each subject began by reading the experiment instructions. The experimenter then reviewed orally with the subject the instructions and answered any questions the subject had regarding the instructions. Next subjects were given the task of debugging a practice program, recording the statements they examined for correctness. During the practice, subjects were encouraged to ask the experimenter questions about experiment procedures that were unclear to them. Also during the practice, the experimenter occasionally prompted subjects to be sure subjects were making a complete record of their debugging activities.

Phase II: Subjects were given the task of debugging a short program, recording as they went the statements they examined for correctness. Before they began debugging, subjects were given a sample input data file and shown both the output generated and the output expected when the program ran using the sample data file. Subjects were not told how many or what types of faults the program contained. Each subject was provided a quiet space in which to work. This space included access to a familiar, on-line platform for Pascal programming.

Phase III: The program debugged in Phase II produced sixteen output values. For phase III, subjects were given the task of constructing from the Phase II program a subprogram that calculated and printed only the two output values of the Phase II program that were incorrect. Subjects were not expected to write out the code for the subprogram, but rather were given the option of circling, underlining, or highlighting statements on a hardcopy of the experiment program code. Although experiment instructions read during Phase I informed the subjects there would be a follow-up task to the debugging task, they were not informed what this task would be until Phase III of the experiment began.

At the end of the subject's session, the subject was urged not to discuss the experiment materials or tasks with the other subjects.

2.3 Materials

For maximum control of slice sizes and intersections, the experimenter wrote the programs used in this experiment. No application domain knowledge was required to understand the programs.

Because all subjects had prior experience coding in Pascal, experiment programs were written in Pascal. Subjects did not have access to the code before their sessions. Two programs were used in the experiment, a practice program and the experiment program. Both programs were written in structured, indented style with no documentation but with descriptive variable names. Neither program contained any subprograms. The practice program was a short, 25-line program that calculated the average and standard deviation of a list of integers read from a file. The fault area of the program could be reduced to the single statement:

```
sum := sum + lcv;
```

with possible correction:

```
sum := sum + list[lcv];
```

The statement was embedded in a single *for* loop.

The experiment program was about 200 lines of Pascal code that calculated a number of descriptive metrics on a file of text read by the program. These metrics included: counting blanks and non-blank characters, vowels, double blanks, double vowels, words, lines, blank lines, the maximum length word, the minimum length word, the maximum number of words per line, the minimum number of words per line, the maximum number of words per sentence and the minimum number of words per sentence. The fault area of this program could be reduced to the single statement:

```
persentence := 0;
```

with possible correction:

```
persentence := 1;
```

The statement was embedded in the fourth of four nested *if* statements.

To facilitate statement referencing, hardcopies of program code with a line number attached to each statement were given to the subjects. In addition to hardcopies of the three program codes, other materials provided the subjects included:

Phase I:

1. Experiment instructions.
2. A copy of the mathematical formulas for average and standard deviation.

Phase II:

1. Access to the school's VAX computer system. The system has a platform for compiling, linking, and running Pascal programs. All subjects were familiar with this environment.
2. A hardcopy listing of an input file that could be used with the experiment program.
3. A printout of the output produced by running the experiment program with the text file described in 2. as input.
4. A printout of the correct output that should be produced by running the experiment program with the text file of 2.
5. A file containing the experiment program.
6. A file containing a copy of the text file of 2.

Phase III:

1. A new copy of the experiment program.

2.4 Hypothesis and Variables

Experiment I tested the hypothesis: Slicers have a better understanding of program code after debugging than non-slicers do. The independent variable was whether the subject was a slicer or a non-slicer. Slicers were determined from the data collected in Phase II of the experiment. Debuggers who stayed mostly within the slice of the incorrect output variable were considered slicers; everyone else was considered a non-slicer. The dependent variable in the experiment was understanding. Understanding was

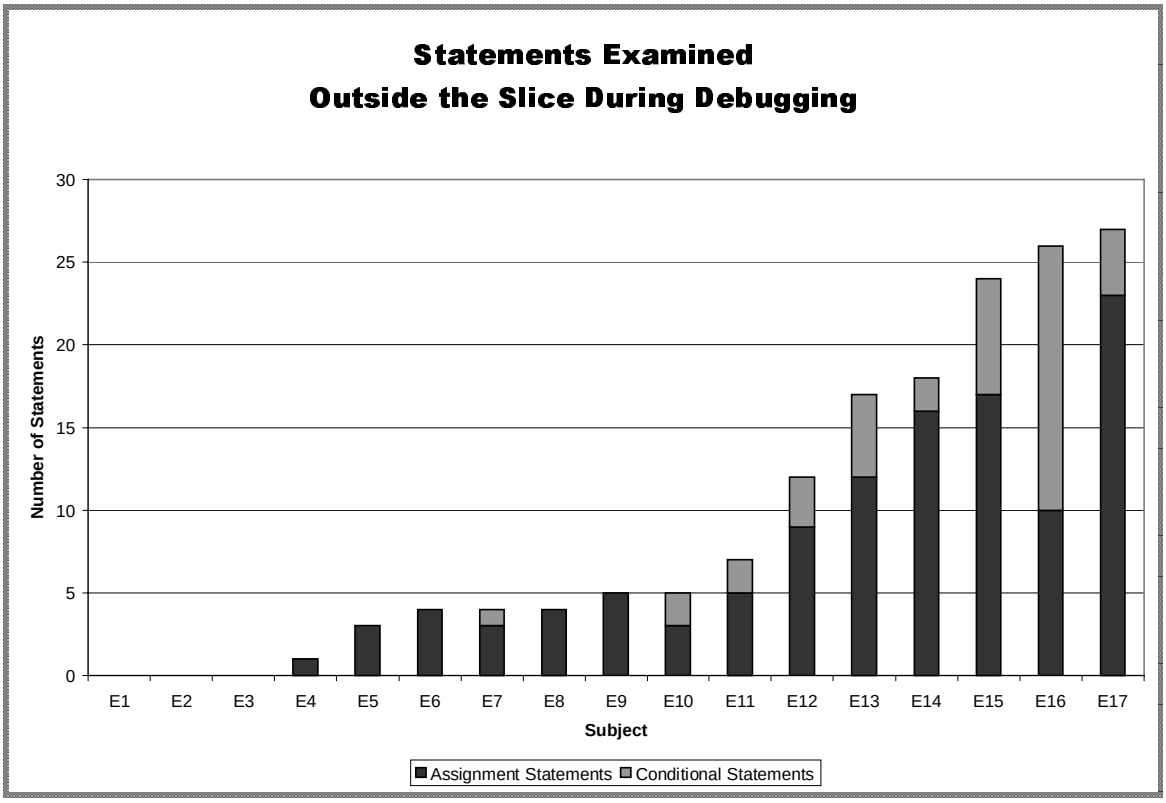

Figure 2

measured by the accuracy of subprogram construction during Phase III of the experiment.

2.5 Data Analysis

Phase II: Due to the small sample sizes of the slicer and non-slicer groups and the moderately skewed distribution of the test variables in Phase II, non-parametric statistical analyses were appropriate for comparing data collected from this phase of the experiment. The Mann-Whitney test was chosen.

Phase III: In Phase III of the experiment, distribution of the test variables was approximately normal, thus parametric statistical analyses were appropriate for comparing data collected. The two-tailed t-test was chosen.

2.6 Results and Conclusions

The data collected from Phase II of the experiment was used to classify debuggers into two groups and to validate that each group used significantly different sets of statements to localize the problem fault area. The latter is an indication that the mental model followed in localizing the fault area was different for the two groups.

During Phase II of the experiment, subjects recorded the statements they examined for correctness and their start and stop times. Figure 2 shows the number of assignment and conditional statements outside the faulty slice each subject examined during debugging. Those subjects who examined one or no statements outside the slice were classified as slicers. All others were classified as non-slicers. Four of the subjects were found to be slicers while the remaining thirteen subjects were found to be non-slicers.

The classification scheme used to label debuggers as slicers or non-slicers ensures that only non-slicers examine statements outside the fault slice area. However, the scheme does not account for the number or the types of statements outside the slice that the non-slicers examined. Even so, not surprisingly, we found that non-slicers examined a significant number of statements outside the faulty slice. One can compare separately the number of assignment statements and the number of control statements the non-slicers looked at outside the slice. Here too the number of statements examined was significant.

- On the average non-slicers examined twelve statements outside the slice while slicers examined

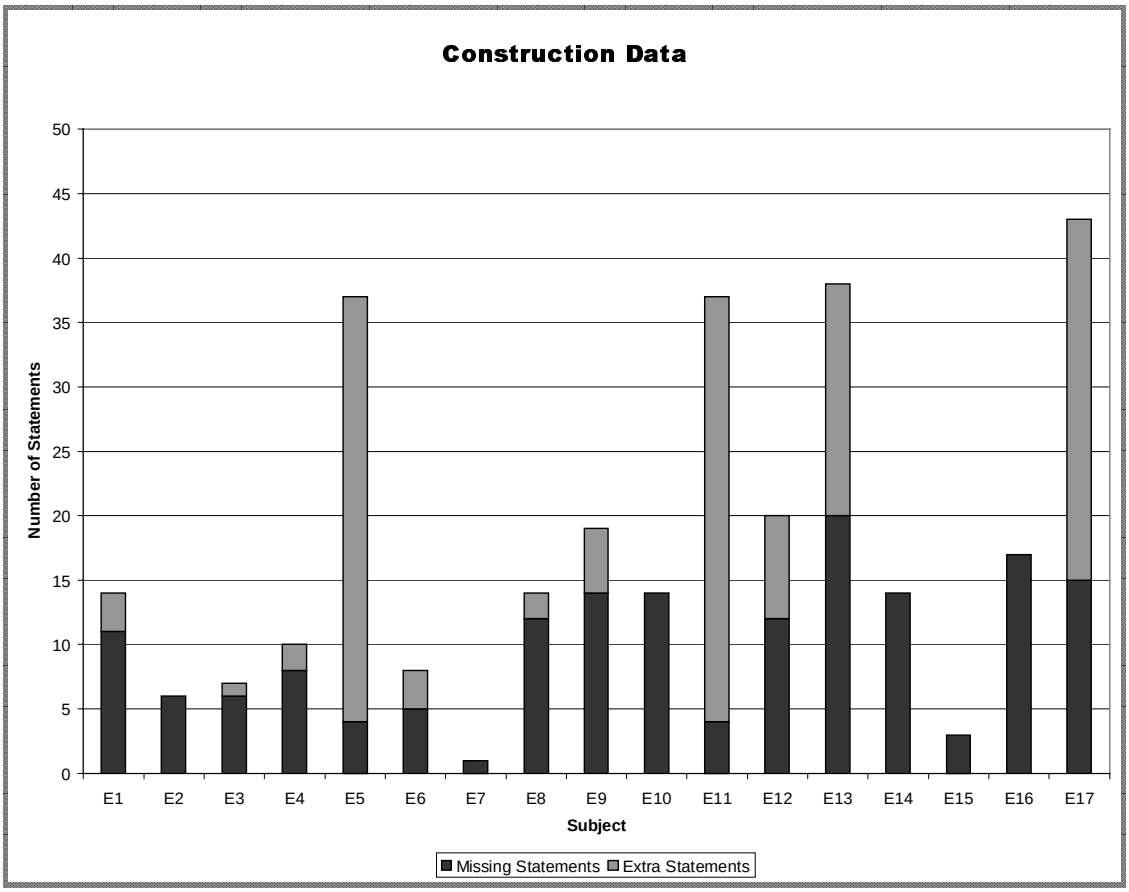

Figure 3

zero statements. This is a statistically significant difference of $p = .001$.

- On the average non-slicers examined nine assignment statements outside the slice while slicers examined zero statements. This is a statistically significant difference of $p = .001$.

- On the average non-slicers examined three conditional statements outside the slice while slicers examined zero statements. This is a statistically significant difference of $p = .045$.

More surprising is the fact that slicers and non-slicers differed significantly on the number of statements they examined inside the faulty slice, which represented about one-third of the program code. Non-slicers actually examined more statements of the faulty slice than did slicers.

- On the average slicers examined nine statements inside the slice while non-slicers examined fourteen statements. This is a statistically significant difference of $p = .0025$.

The average time the slicers took to debug code was significantly less than the time taken by non-slicers.

- On the average slicers took fifteen minutes to debug the program while non-slicers took thirty-three minutes. This is a statistically significant difference of $p = .045$.

In the experiment, subjects classified as slicers limited their debugging activities to code inside the slice of the incorrect output values while non-slicers did not. This seems to imply slicers gain an understanding of the faulty code quicker than non-slicers do. However in order to identify the fault area and make an adequate correction to it one can assume that all debuggers develop, sometime during the

debugging process, an understanding of the faulty code. The purpose of the experiment was to determine if at the end of the debugging process slicers have a better understanding of the faulty code than non-slicers do. The data collected from Phase III of the experiment was used to determine the debugger's level of understanding of the code at the end of the debugging process.

During Phase III of the experiment each subject underlined or highlighted or circled those lines of the original code that were needed to produce a subprogram that would calculate the maximum and minimum words per sentence of the text being analyzed in the faulty program of Phase II. Figure 3 shows the number of statements each subject did not include in the subprogram as well as the number of extra statements that were included.

Slicers differed significantly from non-slicers in their accuracy during the subprogram construction task of the experiment. Inaccuracy was based on both those statements that were missing and those statements that were unnecessary. Again we compare total statements, and assignment and control statements separately.

- On the average slicers erred on nine statements in constructing the subprogram while non-slicers erred on twenty statements. This is a statistically significant difference of $p = .02$.

- On the average slicers erred on six assignment statements in constructing the subprogram while non-slicers erred on thirteen. This is a statistically significant difference of $p = .03$.

- On the average slicers erred on four control statements in constructing the subprogram while non-slicers erred on eight. This is a statistically significant difference of $p = .02$.

In Phase III of the experiment, one would expect time to be directly proportional to accuracy, and, in fact, slicers, who were more accurate than non-slicers in their constructions, did take more time on the average than non-slicers to complete this phase of the experiment. Yet despite this, the average time used by the slicers was not significantly higher than the average time used by the non-slicers.

- On the average slicers took twenty-four minutes to construct the subprogram of Phase III while non-slicers on the average took eleven minutes. This is not a statistically significant difference ($p = .16$).

The experiment tested the hypothesis that slicers have a better understanding of program code after debugging than non-slicers, where understanding was measured through accuracy during subprogram construction. The data gathered from the experiment supports this hypothesis. After debugging a program slicers were significantly more accurate in constructing a subprogram that isolated the faulty portion of the program than non-slicers were. This leads us to conclude that a relationship between slicing while debugging and program understanding exists. It does not however demonstrate that slicing leads to or causes either improved debugging or understanding. Experiment II was designed to investigate part of this question.

3 EXPERIMENT II

Experiment II was conducted to test the hypothesis that slicing while debugging improves a debugger's ability to localize the program fault area. For the experiment, subjects were divided into two groups, a control group and an experimental group. Both groups were asked to debug two programs, recording as they went the program statements they examined for correctness. Between debugging the first and second program, the experimental group received training on how to use slicing in debugging. Besides the statement lists generated during debugging, start and stop times were recorded for both debugging activities. The experiment data was used to investigate the hypothesis that debuggers are better able to localize the program fault area if they use slicing while debugging, than if they debug without slicing**.**

3.1 Subjects

The subjects were twenty undergraduate students at a small liberal arts college in the Southeastern United States who were enrolled, at the time of the experiment, in a standard first programming course. Each volunteered for the experiment. None had had a previous programming course, but each had completed all but one week of the current course. Before the experiment, none of the subjects had been

exposed to the concepts of slicing or dicing. All subjects received extra credit in the lab portion of the programming course for their participation in the experiment. The same instructor taught all the lecture and lab sections of the programming course, from which the subjects were recruited. The course used a single programming language, C++, and had weekly instructor-supervised lab sessions.

3.2 Procedure

The subjects were divided into two groups: Group I, the control group, and Group II, the experimental group. This division paralleled the division into laboratory sections in the programming course from which the subjects were recruited. Each group participated in two debugging sessions held one week apart. The subjects in Group II, the experiment group, also participated in a third session used for training. The training session was held between the two debugging sessions. No time limits were imposed on either of the debugging activities. Subjects spent anywhere from ten minutes to two and a half hours debugging programs. The experimenter was available to the subjects throughout all sessions.

The first debugging session consisted of two phases: an instructions-and-practice phase and a debugging phase. Except that it was held for a group rather than one-on-one, the instructions-and-practice phase of the session mirrored Phase I of Experiment I. Each subject began by reading the experiment instructions; next they were given the task of debugging a practice program, recording the statements they examined for correctness. Throughout, the subjects had the opportunity to question the experimenter regarding any experiment procedures that were unclear to them. The experimenter orally reviewed the experiment instructions with the group after they had read them but before the debugging practice began. Several times during the debugging practice, the group was prompted by the experimenter to record the statements they were examining for correctness.

The procedures followed during the debugging phase of the first session and the second session of the experiment mirrored the procedures followed in Phase II of Experiment I. Subjects were given the task of debugging a program, recording as they went the statements they examined for correctness. Before they began debugging, subjects were given sample input and output data and the program failures (incorrect output) were pointed out to them. Subjects were not told how many or what types of faults the program contained. The subjects worked in the same space they used for the instructor-supervised lab sessions of their programming course. This space included access to an on-line platform for C++ programming. Although subjects all worked in a same room during the experiment, no equipment or materials were shared nor was talking among subjects permitted.

During the training session, the Group II subjects were introduced to the concepts of slicing and dicing and were shown how slicing and dicing can be used in debugging. During this session, subjects were given the task of debugging the program used in the first debugging session. The subjects were urged to use slicing and dicing to complete this exercise.

3.3 Materials

As in Experiment I, the experimenter wrote the programs used in this experiment. Again, this was done for maximum control of slice size and intersections. Also since the experiment subjects were novice programmers, in this experiment, the type of C++ statements used in the code had to be limited. The application domains of the programs were again chosen because they required minimal background knowledge.

Since C++ was the language being used in the programming course in which the subjects were enrolled during the experiment, the three programs used in the experiment were written in it. Subjects did not have access to any of the code before the session in which it was used. Both groups used a practice program and two programs for the debugging activities. All programs were written in structured, indented style with no documentation but with descriptive variable names. None contained any subprograms.

The practice program was a 25-line program that calculated the cost of a long-distance call. The fault area of the program could be reduced to the single statement:

```
time_talked = time2 – time1;
```

with possible correction:

```
if (day1 == day2)
      time_talked = time2 – time1;
else
      time_talked = time_in_day – time1 + time2;
```

The program used for the first debugging session of the experiment was about 100 lines of C++ code

that calculated a number of descriptive metrics about a list of integers read interactively by the program. These metrics included: the minimum of the even integers, the maximum of the odd integers, the average of the positive integers, the number of negative numbers input, the number of integers divisible by ten, and the number of integers divisible by one-hundred. The fault area of the program could be reduced to the single statement:

```
sum = sum + lcv;
```

with possible correction:

```
sum = sum + list[lcv];
```

The statement was embedded in the *else* clause of an *if* statement which was part of a single *for* loop.

The program used in the second debugging session of the experiment was a translation to C++ of the Pascal program used in Phase II of Experiment I. Recall it calculated a number of descriptive metrics on a file of text. The metrics included: counting blanks and non-blank characters, vowels, double blanks, double vowels, words, lines, blank lines, the maximum length word, the minimum length word, the maximum number of words per line, the minimum number of words per line, the maximum number of words per sentence, and the minimum number of words per sentence. The fault area of the C++ version of the program could be reduced to the single statement:

```
persentence = 0;
```

with possible correction:

```
persentence = 1;
```

The statement was embedded in the fourth of four nested *if* statements.

Although both programs used for debugging in the experiment calculated a number of descriptive metrics, the two programs did not have similar structures. The program that calculated integer metrics began by reading the complete list of integers; storing them in an array as it read. Several disjoint loops were then used to calculate the various metrics from the array. In the second program, each data item was processed as it was read. Here all looping was nested. Also many complex, nested conditional statements were used.

To facilitate statement referencing, hardcopies of program code with a line number attached to each statement were given to the subjects.

In addition to hardcopies of the three program codes, materials provided the subjects included:

General:

1. Experiment instructions.
2. Access to the school's UNIX system. The system has a platform for compiling and running C++ programs. All subjects were familiar with this environment.

For each program, subjects were asked to debug:

1. A hardcopy of a sample input that could be used with the program.
2. A printout of the output produced by running the program with the sample input.
3. A printout of the correct output that should have been produced by running the program with the sample input.
4. A file containing the program.
5. A file containing a copy of the sample input of 1).

3.4 Hypothesis and Variables

Experiment II tested the hypothesis: Debuggers are better able to localize the program fault area when using slicing while debugging than while debugging without slicing.

The independent variable was whether the subject used slicing in debugging or not. This group corresponds exactly to the group of subjects that were trained to use slicing while debugging. Recall from Section 3.1 that subjects in both the trained group and the control group had similar backgrounds and programming experience. Further, the debugging abilities of both groups were measured before the experimental group was trained and shown to be not significantly different.

The dependent variable in the experiment was ability to localize the program fault area. This ability

was measured by the amount of time it took to debug the second program.

3.5 Data Analysis

For both debugging activities the distribution of the test variables was approximately normal, thus parametric statistical analyses were appropriate for comparing the data collected. The two-tailed t-test was chosen. Recall that the same situation existed in Phase III of Experiment I.

3.6 Results and Conclusions

During the first debugging session of the experiment, subjects debugged a program recording the statements they examined for correctness and their start and stop times. Table 2 displays the average debugging time taken by each group for this activity. The average time for the two groups does not differ significantly. This is not surprising since the pool of possible participants was limited to individuals known to have similar backgrounds and experience.

- On the average Group I subjects took 31 minutes to debug the first debugging exercise of the experiment while Group II subjects took 19 minutes. This is not a statistically significant difference, p =.343

To determine if fault localization ability was improved when slicing was added to the debugging process both a control and experimental group were needed. For purposes of comparison, the control group could not contain subjects who sliced naturally. The lists of statements examined during the debugging of the programs were used to eliminate any experiment participant from Group I, the control group, who was a natural slicer. The data showed that one subject fit into this category. The statement lists were also checked and used to confirm that the Group II subjects actually used slicing as part of the fault localization process during the second debugging session and that Group I subjects did not.

The second debugging session of the experiment occurred after the training session on how to use slicing in debugging. As in the first debugging session, subjects debugged a program recording the statements they examined for correctness and their start and stop times. Table 2 displays the average debugging time taken by each group for this activity. The average time of the two groups differs significantly. The experimental group, the group that were trained on how to slice while debugging and actually used slicing in the second debugging exercise, took significantly less time to debug on the average than the control group who debugged without slicing.

- On the average the control subjects (i.e. the non-slicers) took 65 minutes to debug the second debugging exercise of the experiment while the experimental subjects, Group II (i.e. the slicers), took 30 minutes. This is a statistically significant difference of p =.019.

Table 2: Mean Debugging Times

	Experiment I	Experiment II
Group I: The control group	31.37	65.29
Group II: The experimental group	19.30	30.16

The experiment tested the hypothesis that slicing while debugging improves a debugger's ability to localize the program fault, where debugging ability was measured by comparing the time it took debuggers who used slicing to debug code with debuggers who sliced without code. The data gathered from the experiment supports this hypothesis. Debuggers who used slicing were significantly better at localizing the fault area of a program than debuggers who debugged without slicing. This leads us to conclude that slicing while debugging improves a debugger's ability to localize the program fault area.

4 DISCUSSION

In this section we will comment on a variety of topics including: experiment issues, the experiment method, and possible future work.

Experiment I: Experiment I did not investigate the question of causality. Further experimentation is needed before one can decide if the relationship between slicing while debugging and program understanding is caused by slicing or by some other influence. However, the fact that the non-slicers examined significantly more of the faulty portion of the code during debugging than slicers did yet still showed significantly less understanding of the relevant code at the conclusion of the debugging process indicates that such an investigation would be of value.

At the onset of Experiment I, it was believed that subprogram construction time could be used to help gauge understanding. Only after the data had been collected did it become apparent that a raw measure of time was inadequate to judge subprogram construction ability. A more meaningful time measure would need to account for the time it takes to correct the inaccuracies made during subprogram construction. One way of doing this might be to combine construction time with an estimate of the time needed to remove the unneeded statements from the constructed subprogram and an estimate of the time needed to add the statements missing from the constructed subprogram. There are several ways to estimate remove time and add times. However, it is hard to defend such estimates as accurate. Questions such as: does it take equal time to analyze each statement, does it take the same amount of time to *undo* as it does to *do*, and what to do about new errors all lead the authors to believe that no valid conclusions can be drawn from the subprogram construction time data collected.

Of the four subjects classified as slicers in Experiment I, two of these were also dicers. That is to say, two did not examine any statements outside the faulty dice. Table 3 shows several statistics from Experiment I, calculated with a separation of slicers who are dicers from slicers who are not dicers. Because of the small number of subjects in the two categories, we do not generalize any of the results.

Table 3: Dicer Information from Experiment I Phase III

	Dicers	Slicers	Non-Slicers
Average number of statements outside slice/dice examined while debugging	0	0	17
Average number of statements outside dice examined while debugging	0	3	21
Average number of inaccuracies during construction phase	6	12	20

Experiment II: In the following discussion, we use the classification of novice (undergraduates), intermediate (graduate students or beginning professionals), and expert programmers (professionals with experience) suggested in *Empirical Studies of Programmers: Fourth Workshop* [3].

As mentioned above, Weiser and Lyle showed, using empirical data, that debuggers who were given a dice to debug took less time than debuggers who were given the original program [9]. The experiment was part of a study on slicing-based debugging tools. This experiment result, although related to it, is not the same as the result shown in the second of our experiments. Our aim was to investigate whether debuggers who find and use slices or dices as part of the fault localization process are better debuggers than those that do not incorporate these reduction methods into their activities. All subjects in Experiment II began with the original program, still the subjects trained in how to use slicing in debugging took less time to localize the fault area than those who were not trained. One should also note that the Weiser/Lyle work used intermediate and expert programmers as subjects, whereas our work used novice programmers. The two results complement each other and in concert provide strong evidence of the value of dicing during debugging.

In his initial work on slicing, Weiser showed [8] that expert programmers naturally use slicing while debugging. The data gathered in Phase II of Experiment I and in Experiment II, discussed above, shows the opposite to be true with novice programmers. Of the seventeen subjects in Experiment, I only four used slicing. Of the twenty subjects in Experiment II, only one was identified as a natural slicer. This is about one-seventh of the combined sample sizes. Yet if slicing leads to improved understanding, slicing may be a valuable tool for novice programmers. However, novice programmers will have to be taught slicing.

The experiment method: Researchers are always searching for effective experimental methods that are non-invasive, easy to learn and which do not alter the subject's normal behavior. The method of recording statement numbers as the corresponding statements are evaluated for correctness, used in this experiment,

proved itself to be a method that meets all three of these criteria. Observation during the training sessions showed that recording statements does not change the subjects' natural patterns of debugging. Also the training period needed to teach subjects how to record which statements they are evaluating for correctness was short yet effective. The time involved in training was about 20 minutes. Lastly, once learned, the method is clearly non-invasive. Subjects do not have to be observed while debugging or prompted in any way.

Future work: The two experiments described in this paper support the premise that slicing while debugging has a positive effect on the debugging process. Experiment I indicates that debuggers who use slicing while debugging have a better understanding of the relevant code at the end of the debugging process than those who do not use slicing. Experiment II indicates that debuggers are better able to localize the program fault area if they use slicing while debugging, than if they debug without slicing. The conclusions mentioned above give rise to several interesting questions, each worthy of further investigation. The most obvious question is:

- Does slicing cause understanding?

But there are several others that merit further study too including:

- Do debuggers slice only to start the fault localization process, or do they use slicing repeatedly throughout the process?
- How much use do debuggers make of the sample program run which produced the program failures during fault localization (i.e. Is the execution used to slice away unexecuted statements during fault localization)?
- Is there a significant difference between the code understanding of debuggers who use dicing and debuggers who just use slicing during the fault localization process?
- Do slicing tools improve the fault localization process?

These questions have ramifications in both computer science education and automated debugging tools. A strategy shown to provide both quick reduction of the code fault area and better understanding of the code at the end of the debugging process is a strategy worth teaching the novice programmer. It may also have value as part of an automated debugging tool.

REFERENCES

[1] Algorithmic and Automatic Debugging Home Page. http://www.cs.nmsu.edu/~mikau/aadebug.html.

[2] *IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 729-1983).* Institute of Electrical and Electronics Engineers. New York, (1983).

[3] Jurgen Koenemann-Belliveau, Thomas G. Moher, and Scott P. Robertson (Eds.). *Empirical Studies of Programmers: Fourth Workshop.* Ablex Publishing, Norwood, NJ.

[4] Jens Krinke. Bibliography, http://brahms.fmi.uni-passau.de/st/staff/krinke/slicing/.

[5] James Lyle. References to Program Slicing. http://hissa.ncsl.nist.gov /~jimmy/refs.html.

[6] Mark Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method.* Ph.D. Thesis, The University of Michigan, (1979).

[7] Mark Weiser. Program Slicing. *Proceedings of the Fifth International Conference on Software Engineering*. 439-449, (1981).

[8] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446-452 (1982).

[9] Mark Weiser and James Lyle. Experiments on Slicing-Based Debugging Aids. *Empirical Studies of Programmers*, Ablex Publishing, Norwood, NJ, 187-197, (1986).