Software Protection against Reverse Engineering Tools

Joshua A. Benson

SOFTWARE PROTECTION AGAINST REVERSE ENGINEERING TOOLS

THESIS

Joshua A. Benson, Captain, USAF
AFIT/GIA/ENG/07-01

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GIA/ENG/07-01

SOFTWARE PROTECTION AGAINST REVERSE ENGINEERING TOOLS

THESIS

Presented to the Faculty

Department of Systems and Engineering Management

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science

Joshua A. Benson, BS

Captain, USAF

March 2007

AFIT/GIA/ENG/07-01

SOFTWARE PROTECTION AGAINST REVERSE ENGINEERING TOOLS

Joshua A. Benson, BS
Captain, USAF

Approved:

/signed/	
Dr. Rusty O. Baldwin (Chairman)	date
/signed/	
Dr. Richard A. Raines (Member)	date
/signed/	
Dr. Paul D. Williams (Member)	date

AFIT/GIA/ENG/07-01

Abstract

Advances in technology have led to the use of simple to use automated debugging tools which can be extremely helpful in troubleshooting problems in code. However, a malicious attacker can use these same tools. Securely designing software and keeping it secure has become extremely difficult. These same easy to use debuggers can be used to bypass security built into software. While the detection of an altered executable file is possible, it is not as easy to prevent alteration in the first place. One way to prevent alteration is through code obfuscation or hiding the true function of software so as to make alteration difficult. This research executes blocks of code in parallel from within a hidden function to obscure functionality.

This method is tested on six programs; a DOS version of the UNIX grep utility and five computational functions: Fast Fourier Transfer, Successive Over-Relaxation, Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization. It tests the impact of using four, eight, and twelve parallel threads of execution to obscure functionality.

The concept is effective, but is limited due to the cost associated with using threads. The computational functions make millions of calls to the hidden function. The average cost per thread for these five functions turns out to be 7.04906 x 10^{-6} seconds. The grep function does not make millions of calls and is therefore more feasible. Care must be taken to ensure the compiler does not remove parallel threads if optimization is used.

Acknowledgments

First and foremost, I would like to thank my wife and children for their support while attending AFIT. I would also like to thank my parents for providing me with so many great opportunities growing up; truly making me the person I am today.

I sincerely appreciate the guidance and support provided by my faculty advisor, Dr. Rusty Baldwin. His insight and experience kept me on the right track. I would also like to thank the Anti-Tamper Software Protection Initiative Office from the Air Force Research Laboratory for sponsoring me in this research.

Joshua A. Benson

Table of Contents

List of Figures

List of Tables

SOFTWARE PROTECTION AGAINST REVERSE ENGINEERING TOOLS

I. Introduction

1.1 Background

There are a multitude of techniques available to protect software. Development of these techniques is largely driven by the financial losses incurred due to copyright violations of digital rights and software piracy. Early defense mechanisms were largely limited to direct media-based protection and serial numbers. These have since evolved into online activations, hardware-based protection, and software as a service [Eli05]. Other techniques include processor dependent code, encryption, and obfuscation [CTL97, CTL98].

Obfuscation with parallel code execution introduces multiple concurrent paths of execution which obscures the true control flow of the program and makes tracing execution with a dynamic disassembler difficult. Parallelization of code can be accomplished via various programming practices. A programmer can manually program the appropriate threads or identify sections of code to be parallelized during the coding process. A compiler then generates the appropriate threads for the sections of code identified by the programmer, relieving the programmer of the burden of keeping track of threads. OpenMP [Ope05] is an example of a standard which supports automatic generation of parallel code.

1.2 Research Goal and Objectives

The goal of this research is to prevent dynamic disassembly of object code. It is hypothesized that it is more difficult for software code to be disassembled after obfuscation. The particular approach to obfuscation is parallelization. Parallelization executes independent blocks of code concurrently leading to multiple paths of execution that will likely be difficult for an analyst or automated program to follow.

1.3 Assumptions/Limitations

An assumption in this research is that a hidden function is executed in a secure section of memory, local to the machine. Accessing this function adds a delay relevant to the size of the function. This delay is simulated to model the overhead of function execution.

The use of OpenMP parallelization limits application of the techniques discussed in this research to multi-processor, shared memory machines.

1.4 Implications

Parallel code execution masks the functionality in an executable file which can be applied to software being developed by the Air Force.

1.5 Preview

Chapter 2 provides relevant background information on obfuscation, debuggers, and current research. Chapter 3 provides the experimental methodology. Chapter 4 provides detailed information on the design, development, and validation of the test system. Chapter 5 provides statistical analysis and results of the experiment. Chapter 6 presents conclusions and recommendations for further research areas.

II. Literature Review

2.1 Chapter Overview

This chapter introduces obfuscation as a means of software protection. It also presents background information relevant to this research. The chapter concludes with a section on current research.

2.2 Obfuscation

Obfuscation is the process of obscuring or confusing [Web96]. Obfuscation of software transforms source or object code such that it is more difficult for a human to comprehend or a debugger to disassemble accurately. The obfuscated code should be functionally equivalent from a user's perspective [Eli05]. The obfuscation process will likely introduce some performance degradation and an increase in size. This should be kept in mind when weighing the cost versus the benefit of incorporating a particular technique during the obfuscation process.

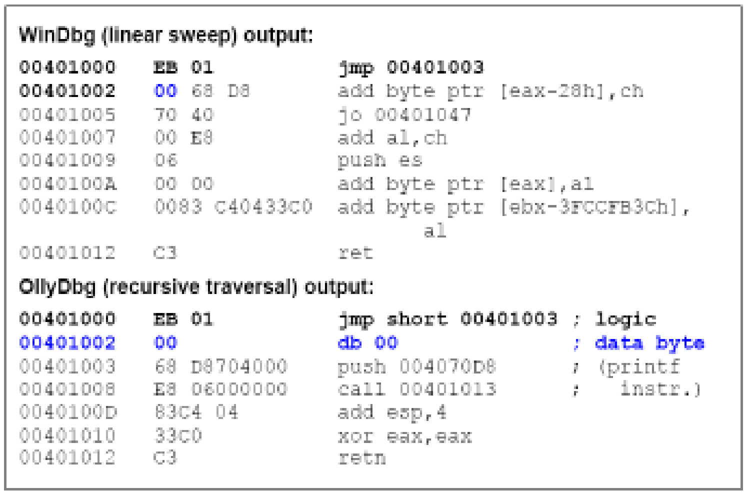
2.3 Debuggers

The operation of debuggers is key to understanding how obfuscation techniques prevent disassembly. Figure 2.1 shows C code with a data byte inserted in the middle of executable code [Dub06]. Many debuggers are not capable of disassembling the object code produced by this code correctly due to the insertion of the data byte and jmp.

```
_asm
{
 jmp L1  ; logic to "skip" data byte
 _emit 0x00          ; inserted data byte
 L1:
}
printf("Hello, World!!!\n");
```

Figure 2.1. Sample Inline Assembly and C code printing "Hello, World!!!" [Dub06]

The disassemblers used by debuggers are implemented in one of two ways: linear sweep or recursive traversal. Figure 2.2 shows the output of these two types of disassemblers after encountering the inserted data byte [Dub06]. WinDbg is a linear sweep disassembler. It goes through an executable line by line assuming everything in the code section is indeed code. This type of disassembler is easy to confuse through code obfuscation. The 00 byte in the example is interpreted as code and combined with the following bytes until it decodes a valid, but incorrect instruction. In Figure 2.2, WinDbg incorrectly produced add byte ptr [eax-28h], ch after the jmp instruction.



WinDbg (linear sweep) output:

```
00401000    EB 01          jmp 00401003
00401002    00 68 D8       add byte ptr [eax-28h],ch
00401005    70 40          jo 00401047
00401007    00 E8          add al,ch
00401009    06             push es
0040100A    00 00          add byte ptr [eax],al
0040100C    0083 C40433C0  add byte ptr [ebx-3FCCFB3Ch],
                                   al
00401012    C3             ret
```

OllyDbg (recursive traversal) output:

```
00401000    EB 01          jmp short 00401003 ; logic
00401002    00             db 00              ; data byte
00401003    68 D8704000    push 004070D8      ; (printf
00401008    E8 06000000    call 00401013      ;   instr.)
0040100D    83C4 04        add esp,4
00401010    33C0           xor eax,eax
00401012    C3             retn
```

Figure 2.2. Disassembly of linear sweep and recursive traversal disassemblers [Dub06]

The other approach is a recursive traversal. This method is much more difficult to confuse, since it follows the control flow of the program. Upon encountering the code in Figure 2.1, a recursive traversal disassembler will follow the jump instruction in the original C code, skipping over the inserted data byte. Once the flow of control is followed to completion, the extra byte is then interpreted correctly as being data. IDA Pro [Ida06] and OllyDbg [Oll05] are recursive traversal disassemblers [Eli05].

Another aspect of disassemblers is the type of analysis they perform on the object code. Analysis can either be static or dynamic. In static disassembly the program being disassembled is not executed, while dynamic disassembly executes the program. The main difference between the two is the amount of time to complete the disassembly. Static disassembly is proportional to the size of the program, while dynamic is a function of the number of executed instructions [LiD03].

2.4 Obfuscation Techniques

Obfuscation techniques can be categorized into four general areas according to the specific target of the transform being implemented: layout obfuscation, data obfuscation, control flow obfuscation, and preventive transformation. Figure 2.3 is a graphical representation of the target of these techniques [CTL97]. Figure 2.4 lists some techniques used in the four areas [CTL97].

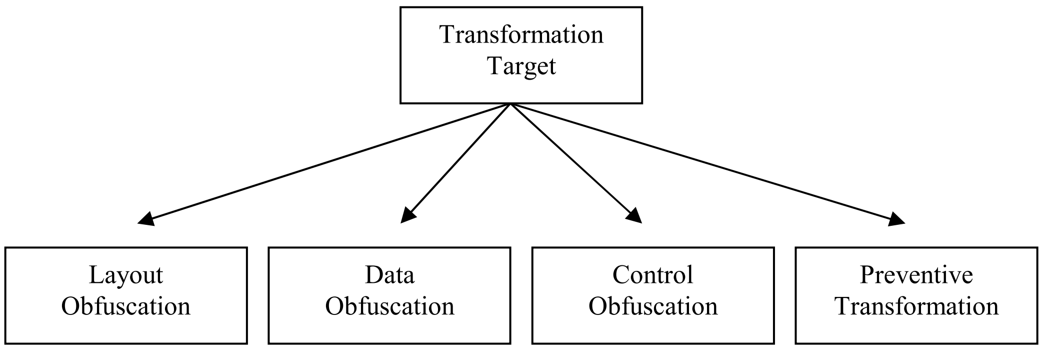


Figure 2.3. Obfuscation Targets [CTL97]

Layout obfuscation makes simple changes to the program including removing the formatting of the program, scrambling variable names, and removing the programmer's comments [CTL97].

Data obfuscation changes the program's use of data or data structures. The storage of data can be obfuscated by replacing current data definitions with those which do not make sense for their intended use. For example, a loop iteration variable can be replaced with another variable type besides an integer. This same principle can be applied to the encoding of data types. Obfuscation via aggregation of data combines scalar variables or changes the structure of arrays. The complexity of obfuscation introduced by the array manipulation operations depends on the particular change being implemented. Splitting and folding arrays is more likely to increase the complexity. However, merging and flattening arrays does not have the same effect, although it does introduce a change in structure. The order in which variables are declared in or the order elements occur in an array can be obfuscated. In some cases, obfuscation includes randomization of declaration order [CTL97].

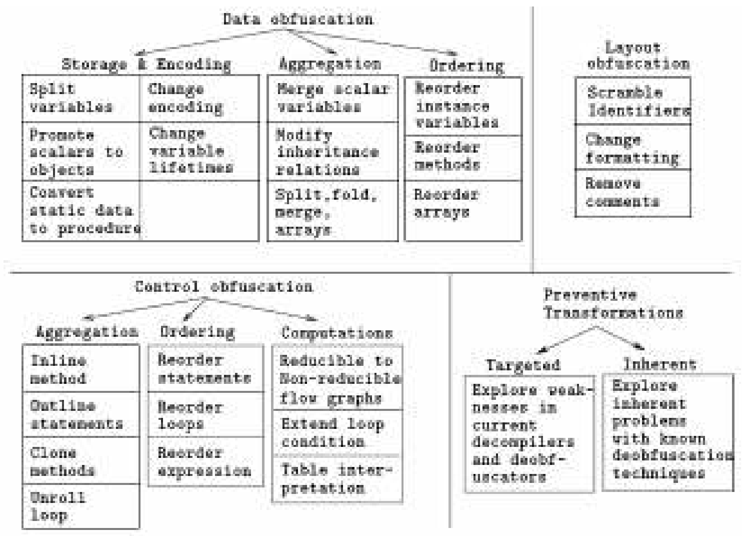


Figure 2.4. Taxonomy of techniques [CTL97]

Control obfuscation changes the flow of the program. Change to control flow can be divided into three separate categories: aggregation transformations, order transformations, and computation transformations [CTL97]. Aggregation transformations remove the program structure which was carefully designed by the programmer to make the code easy to follow and understand. Thus, this transformation removes the high-level organization which once existed. Order transformations simply randomize the order of instructions in the program. Computation transformations remove the original control flow by adding new blocks of code [Eli05].

Opaque predicates can be used to obscure control flow. Opaque predicates are deterministically known to the obfuscator, but are extremely difficult to determine after obfuscation. Opaque predicates introduce what appears to the disassembler to be an undetermined path of execution [CTL97]. A trivial example is an if-then-else statement where the conditional is if (1==2). The true path leads to unreachable code, which is never taken. While the false path is always taken [Eli05].

Executing code in parallel also obscures the control flow. There are two approaches to parallelizing code. The first approach is to insert new functions. These new functions do nothing relevant to the program, but mislead the disassembler while executing concurrently. The second approach divides the program into blocks of code which have no data dependencies between blocks. These blocks are executed concurrently leading to multiple paths of execution. This technique has been shown to increase the number of execution paths exponentially during static analysis [CTL97].

Preventive transformations introduce changes which thwart automated tools attempting to disassemble or deobfuscate the program. These transformations can be inherent or targeted. Simply reordering a loop to be performed backwards is not sophisticated enough to be considered an example of an inherent preventive transformation. However, adding phantom variables which prevent a deobfuscator from reproducing the correct forward version of the loop is a preventive transformation [CTL97, Eli05].

2.5 Measurement of Obfuscation

The level of code obfuscation can be measured using a combination of four metrics. The first metric is potency. Potency measures how well obfuscation techniques obscure the original program. McCabe and Harrison metrics are typically used to measure the complexity of a program. The presumption is that as complexity rises, so does the level of obfuscation [CTL98].

The second metric is resilience. Resilience measures how well a program will stand up against attacks from an automated program. This metric combines two factors: the amount of time required for a programmer to design and implement the automated program, and the amount of time and memory required by the program to perform the attack [CTL98].

The third metric is stealth. Stealth is the ability to hide from an analyst. Large sections of code written in different styles or introducing large extraneous numbers for the purpose of opaque variable calculations will draw the attention of an analyst [CTL98].

The fourth element is the cost. Cost includes the delay in execution time and the increase in program size [CTL98].

2.6 Concurrency Techniques

Parallel code execution introduces multiple concurrent paths of execution. Concurrency obscures the true control flow of the program such that it is difficult for a dynamic disassembler to reconstruct the original correctly.

2.6.1 Compiler Optimizations

Code optimization is carried out by compilers to decrease execution time. However, the functionality of the program must not be changed during the optimization process, otherwise the intent of the programmer is not preserved. Figure 2.5 is an example optimization process carried out by certain high performance compilers [Wol96].

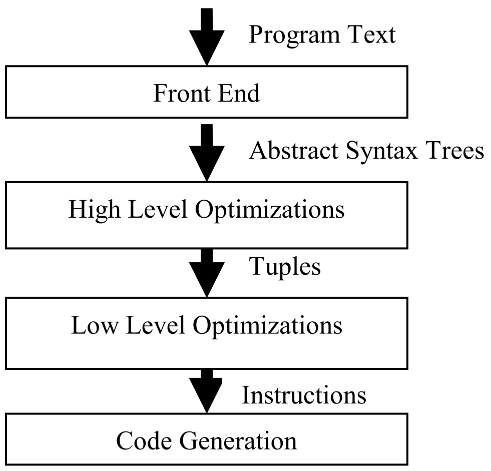


Figure 2.5. Structure of a high performance compiler [Wol96]

When optimizing for parallel execution, each block of code needs to be optimized, not just the original program otherwise the program execution time will be limited by the unoptimized blocks. A high performance compiler uses several phases to optimize blocks of code. A standard front end for compilers will immediately transform the program into a representation that does not retain the high-level structure of the program. Concurrency, however, requires the high level structure be maintained for further analysis in later phases of optimization. The front end of the high performance compiler of Figure 2.5, for example, produces abstract syntax trees in the high level optimization phase to use during its generation of pairs of ordered objects or tuples. The low level optimization phase uses the tuples, along with the details of the machine (the number of processors, the instruction pipeline, or the general architecture) to produce the appropriate instruction set for code generation [Wol96].

2.6.2 Data Dependency

Given enough processors, data with no dependencies would allow an entire program to be executed concurrently. Since such independence is unrealistic, data dependencies need to be determined. Figure 2.6 [Wol96] provides a simple example of data dependency.

S_1: A=0
S_2: B=A
S_3: C=A + D
S_4: D=2

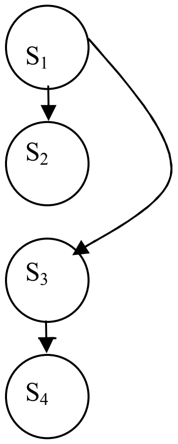


Figure 2.6. Sample program with data dependence graph [Wol96]

In this example S_2 is dependent on S_1, since it uses A. If S_2 were to be performed before or concurrently with S_1, A's value could be wrong. This is an example of flow dependence. Flow dependence occurs when a value is assigned and also used in a later statement. S_3 must occur before S_4, since D is being reassigned in S_4. This is an example of anti-dependence. Anti-dependence occurs when a value is used and then changed in a later statement. For S_2 and S_3, however, S_2 can be executed before S_3, S_3 can be executed before S_2, or they can be executed concurrently. The three cases for S_2 and S_3 hold as long as S_1 is executed first. There is another type of dependence not illustrated in the example called output dependence. Output dependence occurs when a value is assigned in one statement and then later reassigned [Wol96].

2.6.3 Conversion of Standard Code to Parallel Code

Achieving parallelization of code can be accomplished via programming practices or automated tools. A programmer can manually program the appropriate threads or identify sections of code to be parallelized during the coding process.

Automated tools relieve the programmer of having to keep track of threads. Modern tools are capable of taking the original code and creating threads automatically. In OpenMP, for example, identifying the section of code to be executed in parallel simply requires inserting the appropriate OpenMP pragmas [GaI05]. However, variables must be examined to determine if they need to be shared or kept private to the thread and declared appropriately.

2.7 Current Research

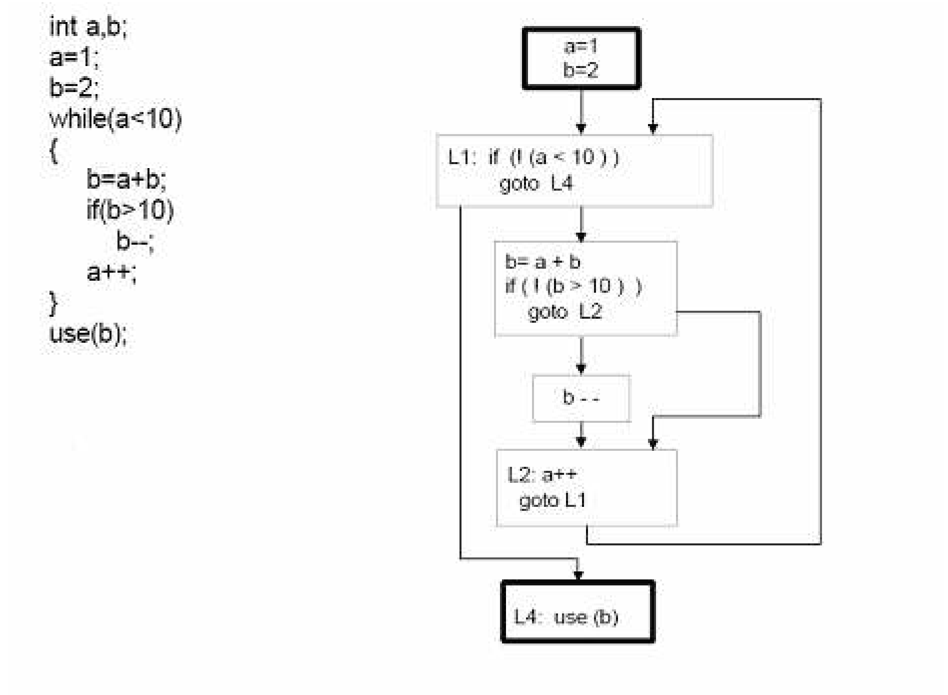
Code obfuscation is the focus of many research efforts. Many of these center on preventing static disassembly. It is instructive to review them to determine how they are related. Disassembly approaches can be categorized based on the type of analysis being conducted.

2.7.1 Evading Static Disassembly

2.7.1.1 Aliases

Static analysis can be prevented by introducing extra pointers called aliases to obscure the control flow. In a scheme designed to disrupt control flow [WHK00], the effectiveness of aliases rest on three architectural elements. The first element is a secure control server. The second is secure network communications between the deployed program and the control server. The third element is regular program communication with the control server to verify its state.

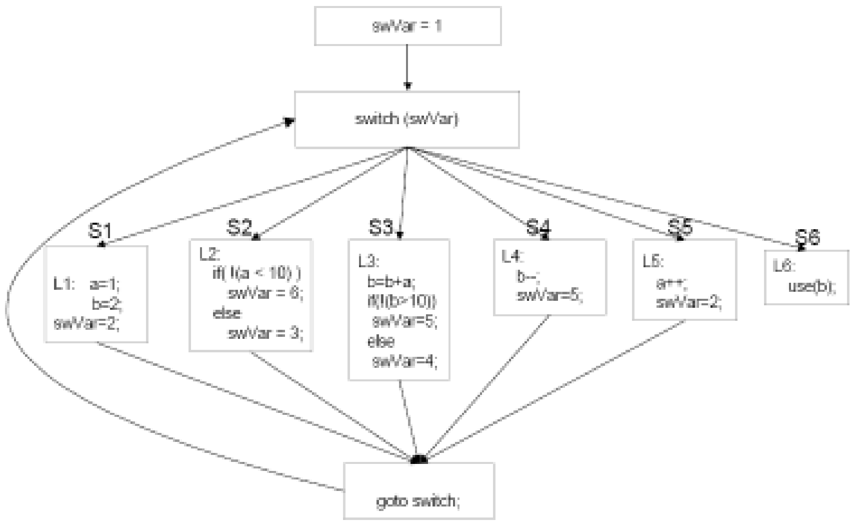
Aliases prevent intelligent tampering and impersonation attacks, while the architectural elements enable a program to perform self-checking and defend against other attacks. Intelligent tampering and impersonation attacks require a detailed analysis of the program. Aliases increase the difficulty of performing the analysis in a three-phased approach. Figure 2.7 shows the first phase, dismantling of high level constructs [WHK00, WDH03]. All high-level language control flow structures (cases, whiles, for-loops) are replaced with an equivalent if-then-goto statement. This creates a flattened representation of the program with data dependencies between branches as shown in Figure 2.8. The global variable *swVar* is used to control the flow. The variable is updated appropriately to maintain the original control flow. For example, S1 first performs the initialization of variables *a* and *b*, then assigns 2 to *swVar*. After returning to the switch, flow will proceed to S2.



```
int a,b;
a=1;
b=2;
while(a<10)
{
     b=a+b;
     if(b>10)
          b--;
     a++;
}
use(b);
```

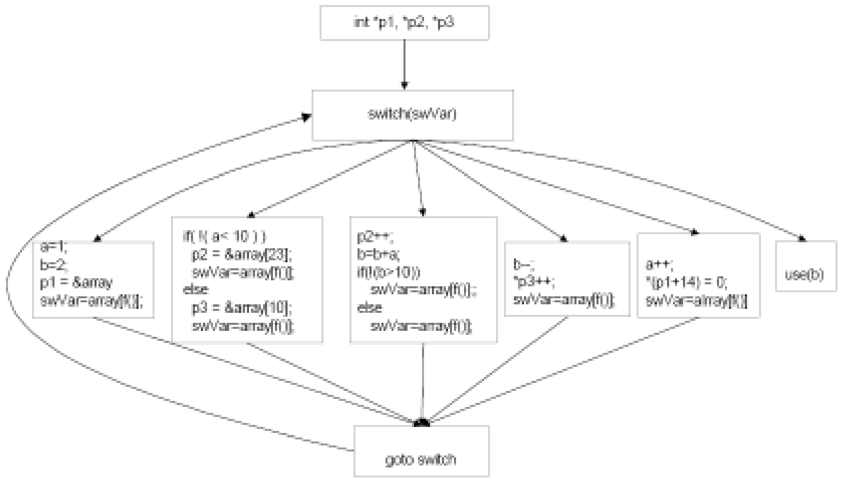
Figure 2.7. Dismantling of high-level constructs [WHK00, WDH03]

The second phase creates a global array in which branches are determined dynamically, instead of the branches being constant values assigned to *swVar* as in Figure 2.8. The third phase adds extra pointers in every function. Figure 2.9 shows the final version of the program after completing the transformation [WHK00, WDH03]. The pointers are assigned through introduced code to valid data variables and global data. All of the original references to the variables are replaced with pointers to include the data dependencies introduced in the first phase. Static analysis of this code will result in the incorrect conclusion that the global array is changing. This increases the number of possible flows of control [WHK00, WDH03].



```
swVar = 1

switch (swVar)

S1
L1:   a=1;
      b=2;
swVar=2;

S2
L2:
  if( !(a < 10) )
     swVar = 6;
  else
     swVar = 3;

S3
L3:
  b=b+a;
  if(!(b>10))
   swVar=5;
  else
   swVar=4;

S4
L4:
  b--;
  swVar=5;

S5
L5:
  a++;
  swVar=2;

S6
L6:
  use(b);

goto switch;
```

Figure 2.8. Transform to indirect control transfers [WHK00, WDH03].



```
int *p1, *p2, *p3

switch(swVar)

a=1;
b=2;
p1 = &array
swVar=array[f()];

if( !( a< 10 ) )
  p2 = &array[23];
  swVar=array[f()];
else
  p3 = &array[10];
  swVar=array[f()];

p2++;
b=b+a;
if(!(b>10))
   swVar=array[f()];
else
   swVar=array[f()];

b--;
*p3++;
swVar=array[f()];

a++;
*(p1+14) = 0;
swVar=array[f()]

use(b)

goto switch
```

Figure 2.9. Completed transform using pointer manipulation [WHK00, WDH03]

2.7.1.2 Junk Byte Insertion, Branch Functions, and Call Conversion

Another obfuscation approach [LiD03] prevents static disassembly by combining several techniques [CTL97]. As implemented, the system is capable of implementing junk byte insertion, branch functions, and call conversion [LiD03]. These three transformations exploit the weaknesses of static disassemblers to determine the return of control flow.

Junk bytes inserted where the disassembler would typically expect to find valid executable code must: (1) be a partial instruction and (2) not be reachable at runtime [LiD03].

Figure 2.10 illustrates the implementation of branch functions [LiD03]. The branching function determines the location to branch to, *b1*, based on the calling location, *a1*. This exploits the assumption that the flow of control will return to the location following the initial function call.

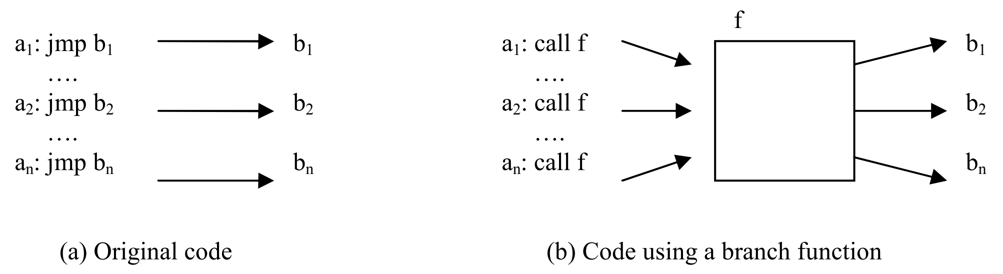


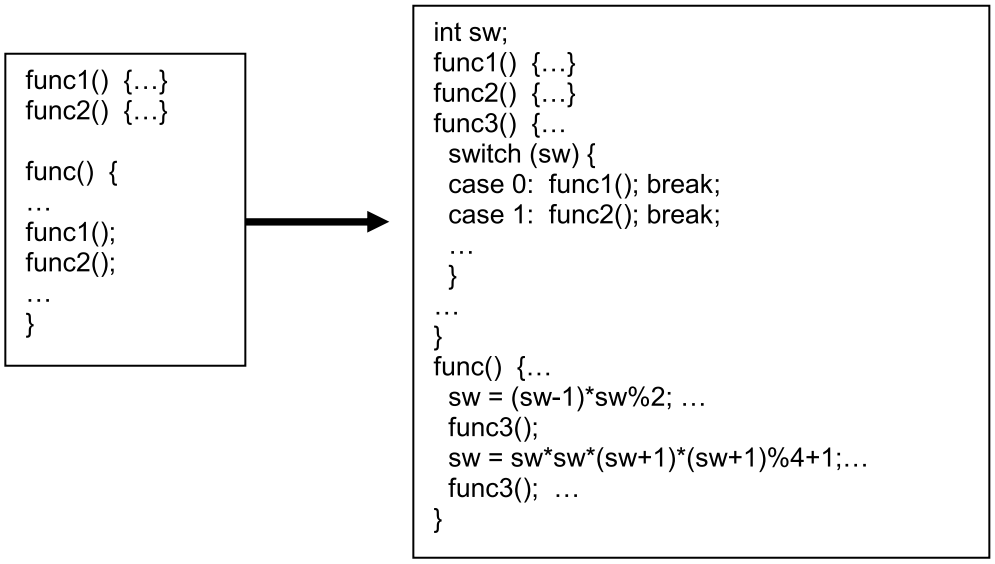
Figure 2.10. Branch functions [LiD03]

Call conversion places junk bytes immediately after function calls, where they would normally be disallowed because they could be reached at runtime. The call is converted in a branching function which branches beyond the inserted junk bytes [LiD03].

2.7.1.3 Obstructing Interprocedural Analysis, Merged Function Calls, and Redundant Return Statements

Obfuscation techniques targeting interprocedural analysis can also obscure intraprocedural analysis. The [OSS03] implementation is very similar to the process used by [WHK00, WDH03] for creating aliases, except it focuses on obscuring flow of control between functions and not just within a function. It also uses a three phased approach. Phase one decomposes functions into smaller functions. Phase two forces the use of function pointers for all function calls. Phase three uses arrays to randomly store function addresses.

Additional techniques for obstructing interprocedural analysis include merging function calls and introducing redundant return statements [OSS03]. Figure 2.11 is an example of merging function calls into one function [OSS03]. Functions with the same return types are selected at random to be included in the new merged function. A position variable is created to maintain the calling position. In the example, *sw* is maintaining this position. The introduction of redundant return statements uses opaque predicates in conditional statements so debuggers may perceive a possible alternate flow of control.



```
func1()  {…}
func2()  {…}

func()  {
…
func1();
func2();
…
}
```

```
int sw;
func1()  {…}
func2()  {…}
func3()  {…
  switch (sw) {
  case 0:  func1(); break;
  case 1:  func2(); break;
  …
  }
…
}
func()  {…
  sw = (sw-1)*sw%2; …
  func3();
  sw = sw*sw*(sw+1)*(sw+1)%4+1;…
  func3();  …
}
```

Figure 2.11. Merge function calls into one call [OSS03]

2.7.1.4 Opaque Constructs via Concurrency

The use of concurrent threads can increase the possible paths of execution making it difficult to perform static analysis. For example, *n* statements in a parallel section can be executed in *n!* ways [CTL97, CoT98, Low98]. When concurrency is combined with a strong opaque predicate, it would require exponential time to determine the true control flow [CTL97, CoT98, Low98]. As implemented in [CTL97, CoT98, Low98], a global data structure is updated by concurrently executing threads. The data structure always contains a deterministic opaque value regardless of the execution order of the threads. Figure 2.12 [CoT98] uses the opaque predicate with the property that $7y^2-1$ will never equal x^2, given any integer x and y. In this example, two threads s and t wakeup occasionally to make updates to the values of the global variables *M.X* and *M.Y*. The threads update the variables with random integers. It does not matter when the opaque predicate (highlighted in the figure) is evaluated because Y-1 will never equal X [CoT98], since X holds the square of an integer.

2.7.2 Evading Dynamic Disassembly

2.7.2.1 Metamorphic Code and Subroutine Reordering

The advanced metamorphic engine in [Dub06] is capable of evading both linear sweep and recursive traversal disassemblers by modifying a program during execution which causes the disassembler to incorrectly perform an opcode shift where it should not at certain points. These so-called morph points are locations where a program would never purposely place an invalid opcode prefix. Thus, the resulting shifted opcode appears believable to the disassembler. A morphing function is used to bypass the intended target of the incorrect opcode shift. Figure 2.13 shows a simple function capable of morphing the return address [Dub06]. The morphing function below changes the return address of its function call while it is on the stack by incrementing it by 2.

```
class S extends Thread {
  public void run() {
    while (true) {
      int R = (int) (Math.random() * 65536);
      M.X = R*R; Thread.sleep(3);
}}
class T extends Thread {
   public void run() {
      while (true) {
         Int R = (int) (Math.random().sleep(2));
          M.Y = 7*R*R; Thread.sleep(2);
          M.X *= M.X; Thread.sleep(5);
}}}
public class M {
  public static int X, Y;
  public static void main(String argv[]) {

     S s = new S(); s.start();
     T t = new T(); t.start();
      if ((Y-1)==X)                              ←Opaque predicate will always evaluate to false
        System.out.println("Bogus code!");
      s.stop(); t.stop();
}}
```

Figure 2.12. Sample java code with Opaque Constructs using Concurrency [CoT98]

```
morphFunction  proc near
                         add byte ptr [esp+0], 2
                         retn
morphFunction            endp
```

Figure 2.13. Simple morphing function [Dub06]

The advanced metamorphic engine also includes subroutine reordering which uses a function manager to shuffle and properly maintain an offset value to add to relative address calls. Global parameter and return variables are also needed to handle inter-function calls properly [Dub06].

2.7.2.2 Dynamic Code Mutation

Dynamic code mutation [MAM05] implements a run-time editing process which maps many different sections of code to the same section of memory. Functions needed at run-time are replaced with templates and any references to the function are replaced with a stub which will call an editing engine. The templates are copies of the original code with random obfuscations to mislead the attacker. The editing engine uses an editing script which has the blueprints for regenerating the function correctly. The editing script contains the location of the functions template, the bytes which require changes, and their correct values. Editing scripts are encrypted using a pseudorandom number generator which has been seeded with an opaque variable [MAM05]. Two separate approaches are implemented. Single pass mutation replaces all functions with separate templates, each with their own editing scripts. Cluster-based mutations locate similar functions and replace them with a standard template. Figure 2.14 shows an example of a cluster based mutation [MAM05]. Each function still has a unique editing script to be used by the editing engine to regenerate the original function.

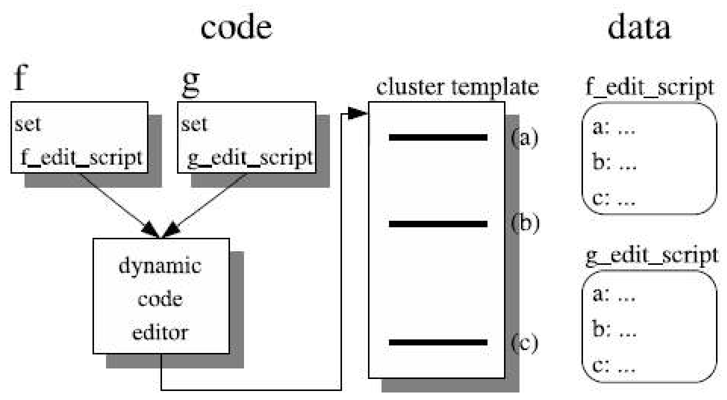


Figure 2.14. Run-Time code mutation with clustering [MAM05]

2.7.2.3 Hiding Program Slices

Hiding program slices [ZhG03] divides function components into two parts, open and hidden. While it is assumed an adversary can tamper with the open components, it is not possible to gain access to the hidden portion, which is located on a secure device, such as a smart card. Figure 2.15 [ZhG03] shows both the mapping and the runtime state of a split function. *S* and *C* represent the state and the code of the open component, while *S'* and *C'* represent that of the hidden component. The state and code required for the two components to interact properly are *s* and *c* respectively.

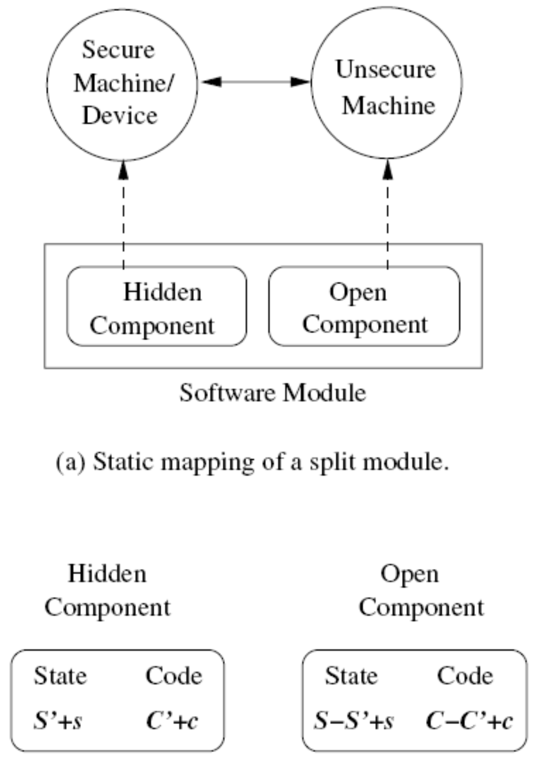


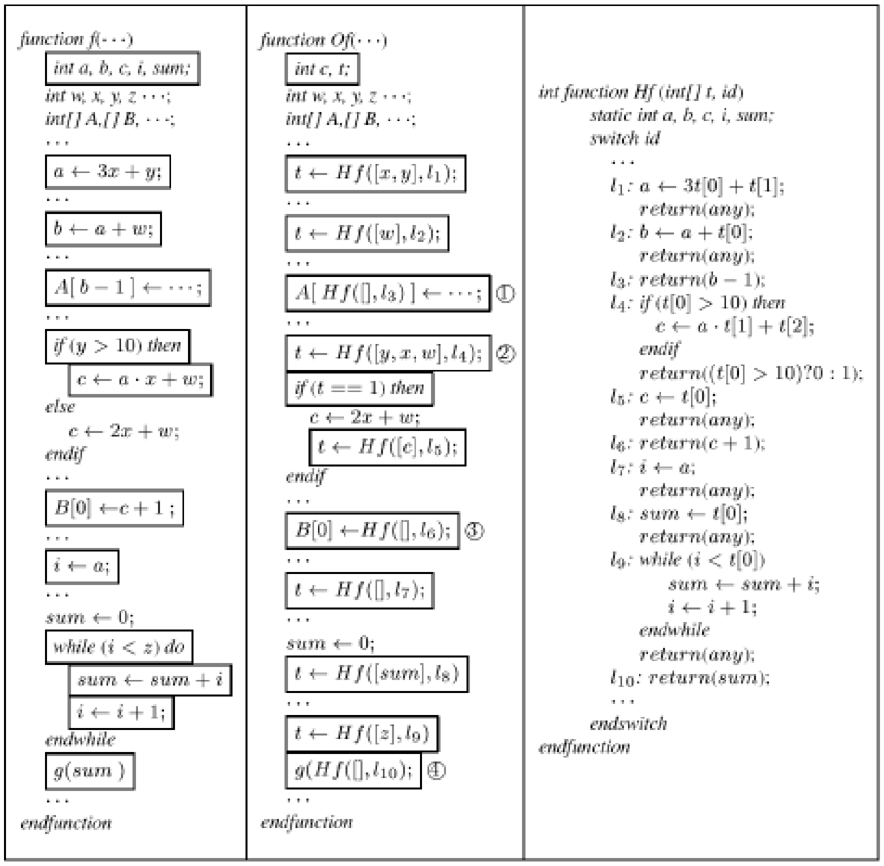
Figure 2.15. Software splitting [ZhG03]

Splitting a function in this manner has two associated costs [ZhG03]. The first cost is the communication delay between the secure device containing the hidden component and the machine containing the open component. The second cost is the computing power of the secure device. To keep both costs down, [ZhG03] implements three restrictions. First, function calls from within a loop are excluded. Second, function calls from within the hidden component are not allowed. Third, only scalar variables local to the function are candidates for moving to the hidden component.

Function splitting begins by selecting a variable to hide and creating a static version in the hidden component. Statements are identified for slicing starting with the one defining the hidden variable. Statements containing the hidden variable or other variables defined at the same time are included in the slice. Next, the remaining variables are analyzed to determine if they can lead to the value of the hidden variable. One of the strengths of this approach is the attacker does not know how many variables are being hidden. Keeping in mind the previous constraints (i.e. function calls, array references), the left and right hand sides of each slice is examined to determine if one side, both sides, or neither side should be include in the hidden component. Next, the remaining statements in the function are examined to determine if they could divulge the existence of hidden variable(s). If so, they are considered for inclusion or partial inclusion in the hidden component.

Figure 2.16 [ZhG03] is an example where *a* is hidden. The mapping of the call to the correct location in the hidden function is contained in the variable *id*. Any values passed to the hidden function are contained in the array *t*. The first column shows the original function with the slices identified in the boxes. The second column shows the new open component with the same boxed statements converted to include calls to the hidden component. The third column shows the hidden component and the corresponding required work. As an example, the second boxed statement in the open component makes a call to the hidden component. It sends the variables *x*, *y*, and the location l_1. The array *t* of the hidden component now contains *x* and *y* as the first two elements and *id* contains l_1. A switch on *id* causes the original work to be accomplished and the result to be stored in the static variable *a*. Any integer can be returned to the open component, since it is not used by any of the statements following it. The circled numbers 1-4 in the second column identify locations where the returned value is used, which could give an attacker useful information. For example, at location 1 the returned value is used to access the array *A*. The authors refer to these points as Information Leak Points (ILP) [ZhG03].

ILPs are analyzed by the authors to determine the complexity of recreating the hidden components associated them. The code corresponding to each ILP is characterized by its arithmetic and control flow complexities [ZhG03]. Figure 2.17 shows the complexities associated with the code for ILP 1 from the example. The arithmetic complexity is determined by the statement's type, inputs, and degree, denoted as <*Type, Inputs, Degree*>. *Type* can be constant, linear, polynomial, rational, or arbitrary. *Inputs* defines the number of variables from the open component which are used by the ILP. *Degree* is the highest degree polynomial when the ILP is not arbitrary. The control flow complexity is determined by the statement's paths, predicates, and flow, denoted as <*Paths, Predicates, Flow*>. *Paths* is defined as either constant or variable. *Predicates* and *Flow* are both defined as either open or hidden [ZhG03].



```
function f(···)
    int a, b, c, i, sum;
    int w, x, y, z ···;
    int[] A,[] B, ···;
    ···
    a ← 3x + y;
    ···
    b ← a + w;
    ···
    A[ b − 1 ] ← ···;
    ···
    if (y > 10) then
        c ← a · x + w;
    else
        c ← 2x + w;
    endif
    ···
    B[0] ←c + 1 ;
    ···
    i ← a;
    ···
    sum ← 0;
    while (i < z) do
        sum ← sum + i
        i ← i + 1;
    endwhile
    g(sum )
    ···
endfunction
```

```
function Of(···)
    int c, t;
    int w, x, y, z ···;
    int[] A,[] B, ···;
    ···
    t ← Hf([x, y], l1);
    ···
    t ← Hf([w], l2);
    ···
    A[ Hf([], l3) ] ← ···;   ①
    ···
    t ← Hf([y, x, w], l4);   ②
    if (t == 1) then
        c ← 2x + w;
        t ← Hf([c], l5);
    endif
    ···
    B[0] ←Hf([], l6);   ③
    ···
    t ← Hf([], l7);
    ···
    sum ← 0;
    t ← Hf([sum], l8)
    ···
    t ← Hf([z], l9)
    g(Hf([], l10);   ④
    ···
endfunction
```

```
int function Hf (int[] t, id)
        static int a, b, c, i, sum;
        switch id
            ···
            l1: a ← 3t[0] + t[1];
                return(any);
            l2: b ← a + t[0];
                return(any);
            l3: return(b − 1);
            l4: if (t[0] > 10) then
                    c ← a · t[1] + t[2];
                endif
                return((t[0] > 10)?0 : 1);
            l5: c ← t[0];
                return(any);
            l6: return(c + 1);
            l7: i ← a;
                return(any);
            l8: sum ← t[0];
                return(any);
            l9: while (i < t[0])
                    sum ← sum + i;
                    i ← i + 1;
                endwhile
                return(any);
            l10: return(sum);
            ···
        endswitch
endfunction
```

Figure 2.16. Splitting of the function *f* initiated with slicing of variable *a* [ZhG03]

$$\boxed{\begin{array}{l} f_{ILP} = b - 1 = a + w - 1 = 3x + y + w - 1 \\ AC(f_{ILP}) = < Linear, 3, 1 > \\ CC(f_{ILP}) = < constant, -, - > \end{array}}$$

Figure 2.17. ILP Arithmetic and Control Flow Complexities [ZhG03]

2.8 Summary

There are several methods available for obfuscating code. Some effectively prevent static disassembly, while others are more robust and can prevent dynamic disassembly. Determining the strength of obfuscation relies on several factors. All obfuscation techniques come with an associated cost.

III. Methodology

3.1 Chapter Overview

This chapter presents the experimental methodology. The system is presented, along with its services, boundaries, parameters, and workload. The factors varied and their associated levels are also presented.

3.2 Experimental Approach

To determine the validity of using parallel threads for security a baseline of the workload application is established to perform code slicing similar to [ZhG03]. The only exception being the hidden function is assumed to be executing from a secure location local to the machine instead of on an external device. Parallel threads are introduced to concurrently execute the sliced code of the hidden function. The number of parallel threads is adjusted to determine the impact of increasing the number of potential execution paths. The execution times of the different levels are analyzed to determine if statistically significant differences are present.

3.3 System Boundaries

The System Under Test (SUT) is the parallelizing system. This system creates obfuscated code by parallelizing the supplied benchmark code. Figure 3.1 shows the system boundaries, parameters, workload, metrics, and expected outcome. The components of the system include the parallelizing tool, the compiler, the debugger, and the Operating System (OS). An OS is a program that manages the computer, including hardware and software. It takes care of many different tasks and coordinates the different elements of the computer [Eli05]. A compiler takes a source file written in a high level language and generates corresponding machine code [Eli05]. A debugger is a program which allows software developers to observe a program during execution [Eli05]. These four components are essential to the system because they are required to produce the obfuscated code. The Component Under Test (CUT) is the tool creating the parallelized code.

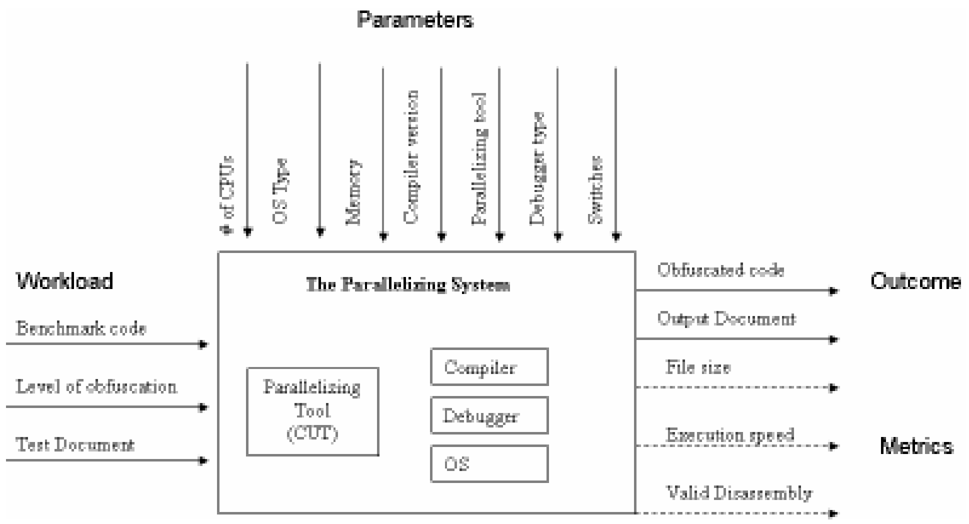


Figure 3.1. System Under Test

3.4 System Services

The parallelizing tool provides a functionally equivalent parallelized version of the source code. OpenMP API [Ope05] constructs implement the parallelized code.

Possible outcomes of the service are:

- the code functions correctly and can be disassembled
- the code functions correctly and cannot be disassembled
- the code does not function correctly and cannot be disassembled
- the code does not function correctly and can be disassembled

3.5 Workload

The system workload is benchmark and open source code. The SciMark2.0 benchmark [Sci2.0] and Ggrep [Gha04] are used. SciMark2.0 measures the performance of common numerical algorithms in scientific and engineering applications. It consists of five computational kernels: Fast Fourier Transfer (FFT), Successive Over-Relaxation (SOR), Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization [PoM04]. Ggrep is a DOS version of the popular UNIX grep utility. It determines if a specified search criteria, a regular expression, matches any of the strings present in the search file [Gha04]. These programs are used because being open source, their availability will make it easier for others to reproduce the experiment if desired.

During execution of the parallelization tool, the system will obfuscate selected functions from SciMark2.0 or Ggrep at a specified level. Level one provides no obfuscation. Level two provides hidden functionality with no parallelization. This is similar to hiding program slices [ZhG03]. Level three provides hidden functionality and parallelization with four threads of execution. Levels four and five add an additional four threads each. Ggrep uses three regular expressions to follow different control paths of execution. The three expressions are: *[n].**, *[I].?*, and *[!ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz].** respectively. A test document to search within is also provided to Ggrep. The parallelization tool produces new source code to run through the compiler to generate an executable file.

3.6 Performance Metrics

Metrics include the file size after obfuscation, the execution speed of the obfuscated file, whether or not the obfuscated file is a functional equivalent of the original, and whether or not the obfuscated file can be disassembled. The newly compiled executable is measured in bytes to determine file size. The stopwatch function of Ggrep starts timing upon entry to the main function and stops timing after searching the test document for a match, just before exiting the program. The stopwatch function also measures the execution time for all five SciMark2 functions. Time is measured in seconds. Functionality of executables is measured in two ways. First, the executables must exit normally including timing information. Second, the output must be the same as the unparallelized versions.

3.7 Parameters

3.7.1 System Parameters

Table 3.1 lists the system parameters of the SUT. The operating system type determines whether or not multi-threading is available. The presence of shared memory to exchange data between multiple processors is required. The number of CPUs in the system determines the number of blocks of code capable of true parallel execution. This parameter also impacts runtime overhead in the OS. The type of debugger drives the strength of the disassembly. The compiler chosen is capable of using OpenMP which provides simplified multi-threading. The optimization level determines the amount of optimization implemented by the compiler. The parallelizing tool is the component under test.

Table 3.1. System Parameters

OS type
Shared Memory
CPUs
Compiler version
Compiler optimization levels
Debugger type
Parallelization tool

3.7.2 Workload Parameters

Workload parameters include the SciMark2.0 and Ggrep source code, the particular function selection, and the test document and expression for Ggrep. The parallelization tool instructs the obfuscation of a particular function.

3.8 Factors

Factors and their associated levels are summarized in Table 3.2. The debugger levels represent the two major disassemblers available. OllyDbg [Oll05] and IDAPro [Ida06] which are both dynamic debuggers. It is important to vary the optimization levels passed to the compiler as optimization can remove the effects of the parallelization tool. A baseline for the benchmark code is also needed. The baseline is established by running the system with the parallelizing tool not in use. All other experiments use the tool.

3.9 Evaluation Technique

A direct measurement of the system is carried out. The system is simple enough to create with components available at AFIT. The system is composed of:

- Microsoft Windows XP Professional Version 5.1.2600 Service Pack 2 Build 2600
- 4.0 GB of RAM
- 2 dual-core processors with hyper-threading (8 Intel Xeon CPU 3.00 GHz)

- Microsoft Visual Studio 2005 Version 8.0.50727.42 [MVS05]
- OllyDbg version 1.10 (dynamic disassembler) [Oll05]
- IDAPro version 4.6.0.809 SPI 32-bit (dynamic disassembler) [Ida06]
- SciMark 2.0 benchmark [Sci2.0]
- Ggrep [Gha04]

The system is validated by determining that it is functional as defined in the performance metrics section, and the validation section of Chapter 4.

Table 3.2. Factors with Associated Levels

	Type of debugger	Compiler switches	Parallelizing Tool	Benchmark Code
Level 1	OllyDbg	optimization on	not being used	SciMark2 FFT
Level 2	IDAPro	optimization off	hidden function in use	SciMark2 LU
Level 3			hidden function with 4 threads of parallelization	SciMark2 Monte
Level 4			hidden function with 8 threads of parallelization	SciMark2 SOR
Level 5			hidden function with 12 threads of parallelization	SciMark2 Sparse
Level 6				Ggrep Expression 1
Level 7				Ggrep Expression 2
Level 8				Ggrep Expression 3

3.10 Experimental Design

A full factorial experiment is conducted. This requires a total of 160 experiments without replications. Table 3.3 summarizes the factors and workload, along with the number of associated levels. Each experiment is replicated 5 times which was sufficient to obtain a width of +/- 10% from the mean at a confidence interval of 90%. This results in a total of 800 experiments performed.

Table 3.3. Experimental Design

	Levels
Debugger type	2
Compiler switches	2
Parallelization tool	5
Benchmark code	8

3.11 Summary

A direct measurement of the system using a full factorial experimental design consisting of 800 experiments is described. The SUT obfuscates various functions in the benchmark code. The SUT collects metrics on file size, execution speed, execution functionality, and whether or not the obfuscated code can be disassembled or not by the debugger present.

IV. System Design, Development, and Validation

4.1 Chapter Overview

The tested system uses Ggrep and SciMark2 benchmark code. This chapter describes the design, development, and validation for the implemented experimental version of Ggrep and the five functions of SciMark2.

4.2 System Design

The system functions at five different levels. Figure 4.1 shows the distinction between levels. Level 1 is the baseline. Level 2 provides code slicing [ZhG03] functionality with a hidden function. Slices of code from the baseline now function within the hidden function. Level 3 provides parallelization of the hidden functionality of Level 2 with 4 threads of execution. Levels 4 and 5 provide the same parallelization of Level 3, except with 8 and 12 threads of execution respectively.

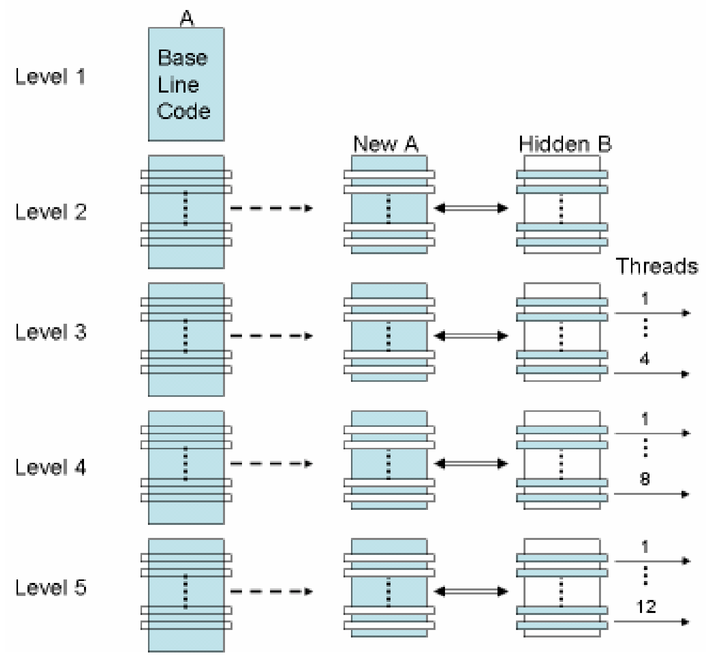


Figure 4.1. System Levels

The system provides simulated security wrapping and unwrapping for the hidden functions of Levels 2 through 5. Before the hidden function is used, bytes of code are decrypted by a simple XOR procedure. When the hidden function is no longer needed, the bytes of code are encrypted by same XOR procedure. This is not necessary for Level 1, since there is no hidden function. This induces some delay relevant to the size of the function to emulate the effect of the hidden function operating in a secure location in memory. It also includes two versions of the Ggrep and SciMark2 executables for each level. The first version has optimization off and the second has it on. The source code for both versions is identical.

4.2.1 Hidden Function Design Details

Slicing of the function [ZhG03] involves selecting portions to be hidden from the viewable function and executing them in a (assumed) secure area via a hidden function. Since the secure area for this experiment is not actually implemented, functions execute locally. However, the design of the hidden function used by the system is similar to that of [ZhG03]. The hidden function requires static variables to maintain values. Referencing of the sliced sections of code requires a call to the hidden function with a location variable. A case statement switches on the location variable to access the appropriate code and the hidden function returns a value for use by the calling function. Limitations set forth by [ZhG03], as described in Section 2.7.2.3, are not adhered to since the hidden function is not on an external device.

4.2.2 Parallelization Design Details

Parallelization of Levels 3 through 5 takes the case statements of Level 2 and turns them into parallelized code. Every case executes simultaneously with a minimum of four cases. Identifying the sections of code to execute in parallel to the compiler simply requires inserting the appropriate OpenMP [GaI05] pragmas. Examination of variables determines if sharing among all threads is necessary or if they can be kept private to a particular thread.

4.3 System Development

This section presents differences between Levels 1 through 3. Levels 4 and 5 are exactly the same as Level 3 except for a global variable change to account for the increased number of threads used.

4.3.1 Ggrep Development Details

Modification of Ggrep [Gha04] for Level 1 is limited to the addition of timing code for metric collection. Timing statistics are collected using the StopWatch in SciMark 2.0 [Sci2.0]. Timing is started after the assert(argc == 3); statement and stopped after exiting the while loop as whown in Figure 4.2. Timing includes Ggrep converting the specified search pattern to a regular expression and comparing it with each word from the test document. Timing is reported prior to exiting the main function.

Level 2 modifications hide elements of the process of creating the regular expression present in the function toRegular. These elements are removed and placed in the function HtoRegular. Figures 4.3 and 4.4 present the Level 2 version of toRegular. Figures 4.5 and 4.6 present HtoRegular. At first glance it would appear to be straight forward to remove the details of toRegular, since the function already contains a case statement. However, a dependency is present in the iteration counter *i*. This is a case of anti-dependence. The variable is used and then updated before its normal single increment. Therefore, the structure trgex of type tstring is created to hold both the regular expression and the current value of the counter.

```
int main(int argc, char *argv[]){
	ifstream inFile(argv[2], ios::in);
	if(!inFile){
		cerr << "Description File is not found!" << endl;
		exit(1);
	}
	cout << "----------" << endl << "--------GREP--------"
<<endl;
	assert(argc == 3);
	Stopwatch Q = new_Stopwatch();
	Stopwatch_start(Q);                  ←Timing Starts
	string rex = toRegular(argv[1]);
	rex = concatExpand(rex);
	IntoPost p(rex);
	rex = p.doTrans();
	NFA nfa = createNFA(rex);
	while(!inFile.eof()){
		string word = "";
		inFile >> word;
		if(process(word,nfa) == true)
			cout << word << endl;
		else;
	}
	Stopwatch_stop(Q);                   ←Timing Stops
	printf("Time to Execute:        %8.8f\n"
,Stopwatch_read(Q));
	Stopwatch_delete(Q);
}//main
```

Figure 4.2. Ggrep.exe [Gha04] Main

```
string toRegular(char * expr)
{
struct tstring tregex;
tregex.iteration=0;
tregex.expression=expr;
string regex = "";  //the resulting regular expression
int i=0;
unsigned char encrypted[2768]={
/*006570:*/ 0x55, 0x8B, 0xEC,
/*006573:*/ 0x6A, 0xFF, 0x68, 0xFD, 0x55, 0x4E, 0x00, 0x64, 0xA1, 0x00, 0x00,
0x00, 0x00, 0x50, 0x81, 0xEC,
                              . . .
                              . . .
/*007023:*/ 0xE8, 0x02, 0xB7, 0xFF, 0xFF, 0x8B, 0xE5, 0x5D, 0xC3};
for (i=0; i<2768; i++)
 encrypted[i]=encrypted[i] ^ 0xFF;
                              . . .
```

Figure 4.3. Ggrep Level 2 toRegular

Figure 4.4 shows two different calls to HtoRegular. Each call is the same except for the location value being passed. The value of *i* is synchronized both before and after the call. The current value of the regular expression is also stored in the variable *regex*, but the details, contained in HtoRegular, for creating the expression remain hidden.

```
                              . . .
                              . . .
//process the entire string
       for(int i = 0; expr[i] != '\0' && expr[i] != '\"';)
       {
              if(!isOper(expr[i]))//if not an operator then a char
              {
                     tregex.iteration=i;
                     tregex=HtoRegular(tregex,0);
                     regex=tregex.expression;
                     i=tregex.iteration;
                     i++;
              }
              else   //it is an operator
              {
                     switch(expr[i])      //a switch on the char read
                     {
                     case '(':
                            tregex.iteration=i;
                            tregex=HtoRegular(tregex,1);
                            regex=tregex.expression;
                            i=tregex.iteration;
                            i++;
                            break;
                            ...
                            ...
                     }
              }
       }
       for (i=0; i<2768; i++)
              encrypted[i]=encrypted[i] ^ 0xFF;
       return regex;       //return the regular expression
}
```

Figure 4.4. Calls to HtoRegular

The function HtoRegular accepts the tstring *z* and the integer *location* as parameters as shown in Figure 4.5 and returns a tstring back to toRegular. Initialization of HtoRegular includes all temporary variables used by elements moved into it, as well as its own static version of the regular expression and the tstring *hiddenstruct*. It uses the variable *firsttime* to complete the work only required on the first call to it, otherwise it copies the regular expression contained in *z* to the static local copy *expression*. Hidden elements are accessed through the *location* variable. The first two cases are shown in Figure 4.6. The entire function contains eight cases.

```
tstring HtoRegular(tstring z, int location)
{
        static string regex="\0";
        stack<char> bracketStack;
        int m=0;
        string tempString2 = "()*";//cannot be part of output
        string tempS = "\0";
        string tempstring="\0";
        bool belong = true;//whether chars belong to the output or not
        int k=0;
        static struct tstring hiddenstruct;
        struct tstring t;
        t.expression="";
        t.iteration=0;
        tempstring=regex;
        static string expression="";
        static int firsttime=1;
        if (firsttime)
        {
                expression=z.expression;
                firsttime=0;
        }
        else z.expression=expression;
                        ...
                        ...
```

Figure 4.5. Initialization of Level 2 HtoRegular

After completion of HtoRegular, it is necessary to simulate the function being wrapped and unwrapped. The functon location is identified in the executable using OllyDbg[Oll05]. To simplify identification, temporary print statements are added to the function. The limits of the entire function from the entry point to the return is identified. The hex digits associated with the opcodes of the function are used to distinguish the function in HexEdit [Hex02]. The hex for the function is copied to an array of unsigned characters (cf., Figure 4.3). This assures the proper length for wrapping and unwrapping the hidden function. Since the goal is to provide delay comparable to the size of the function, a simple XOR with 0xFF for each character in the array is performed just after initialization of toRegular and just prior to exiting toRegular (cf., Figure 4.4). This same process is used for the remainder of the levels and functions requiring wrapping and unwrapping throughout the system.

```
                      ...
                      ...
switch (location){
case 0:       t=z;
              tempstring=regex;
              tempstring += bracket;     //add a bracket
              tempstring += t.expression[t.iteration];//the char
              tempstring += cBracket;    //enclose the bracket
              t.expression=tempstring;
              hiddenstruct=t;
              break;
case 1:       t=z;
              tempstring=regex;
              tempstring += bracket;
              t.expression=tempstring;
              hiddenstruct=t;
              break;
case 2:
                      ...
                      ...

}//switch
regex=t.expression;
return t;
}
```

Figure 4.6. Hidden details of Level 2 HtoRegular

Level 3 modifications transform the work of HtoRegular from Level 2 as in Figure 4.6, to a version executing in parallel as shown in Figure 4.7. The function toRegular remains unchanged. Since Level 3 limits parallelization to four threads, half of the eight cases execute simultaneously. When one finishes, the next section starts. This continues until all of the sections finish. The *#pragma omp parallel sections* indicate the sections of code to be executed in parallel using the number of threads specified by the global variable *Num_of_Threads_to_Use*. The sections themselves are identified by the *#pragma omp section* declarations. It would be possible to introduce threads that do not implement any functionality of the original function by simply adding additional sections.

However, this was not accomplished for this example. Each thread has its own private version of variable *tempstring*. This assures any work done for the real thread of execution is not corrupted. The final version of the tstring *t* is stored in the shared array *holding*. Indirect mapping ensures no correlation between the *location* parameter of HtoRegular and the array element the thread is saving to.

```
                        ...
                        ...
#pragma omp parallel sections firstprivate(regex,
z)private(tempstring,tempS,t)shared(holding)num_threads(Num_Threads_to_Use)
{
        #pragma omp section
        {
                t=z;
                tempstring=regex;
                tempstring += bracket;     //add a bracket
                tempstring += t.expression[t.iteration];//the char
                tempstring += cBracket;    //enclose the bracket
                t.expression=tempstring;
                holding[4]=t;
        }
        #pragma omp section
        {
                t=z;
                tempstring=regex;
                tempstring += bracket;
                holding[3]=t;
        }
                        ...
                        ...
```

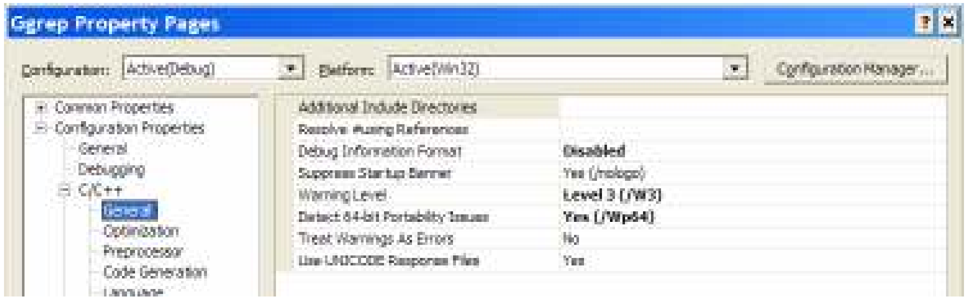
Figure 4.7. Parallelization in Level 3 HtoRegular

Figure 4.8 is the remapping of the array to the static variables via a switch on *location*. The tstring being held is returned to toRegular.

```
                        ...
                        ...
switch (location){
        case 0:         regex=holding[4].expression;
                        return holding[4];
        case 1:         regex=holding[3].expression;
                        return holding[3];
                        ...
                        ...
        }//switch
}
```

Figure 4.8. Remapping in Level 3 HtoRegular

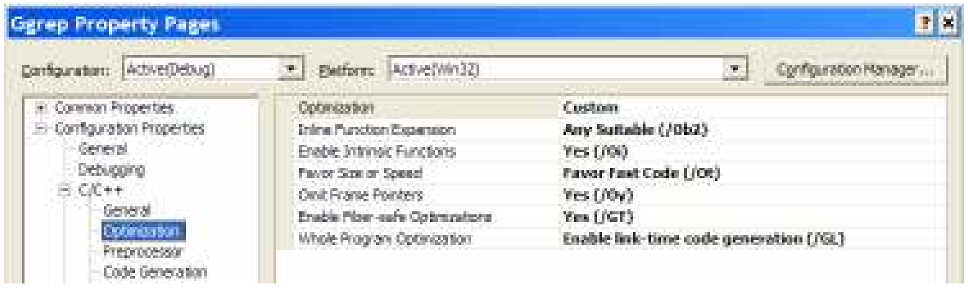
Ggrep with optimization turned on contains the property settings present in Figures 4.9 through 4.11. Changes made from the default program in Visual C++ [MVS05] to the General and Code Generation tabs are driven by incompatibilities with the optimization settings established in Figure 4.10 and incorrect results produced by test runs of Ggrep.



Configuration: Active(Debug) Platform: Active(Win32)

Additional Include Directories	
Resolve #using References	
Debug Information Format	**Disabled**
Suppress Startup Banner	Yes (/nologo)
Warning Level	**Level 3 (/W3)**
Detect 64-bit Portability Issues	**Yes (/Wp64)**
Treat Warnings As Errors	No
Use UNICODE Response Files	Yes

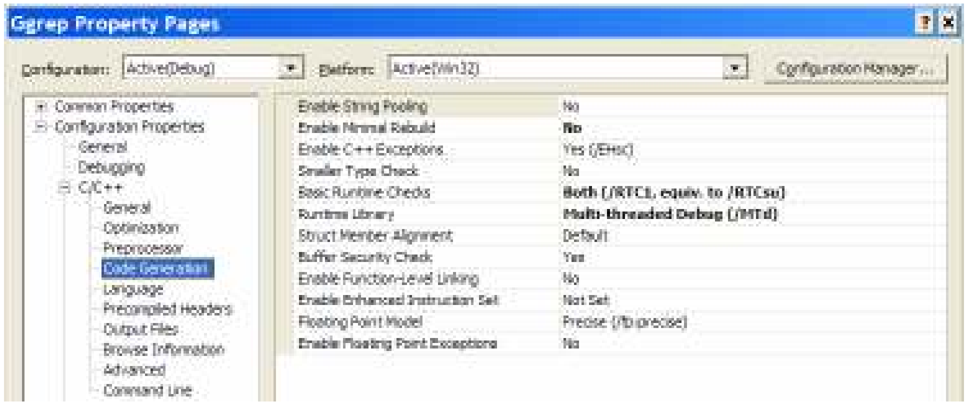
Figure 4.9. General Tab of Ggrep Optimization-On



Configuration: Active(Debug) Platform: Active(Win32)

Optimization	Custom
Inline Function Expansion	**Any Suitable (/Ob2)**
Enable Intrinsic Functions	**Yes (/Oi)**
Favor Size or Speed	**Favor Fast Code (/Ot)**
Omit Frame Pointers	**Yes (/Oy)**
Enable Fiber-safe Optimizations	**Yes (/GT)**
Whole Program Optimization	**Enable link-time code generation (/GL)**

Figure 4.10. Optimization Tab of Ggrep Optimization On

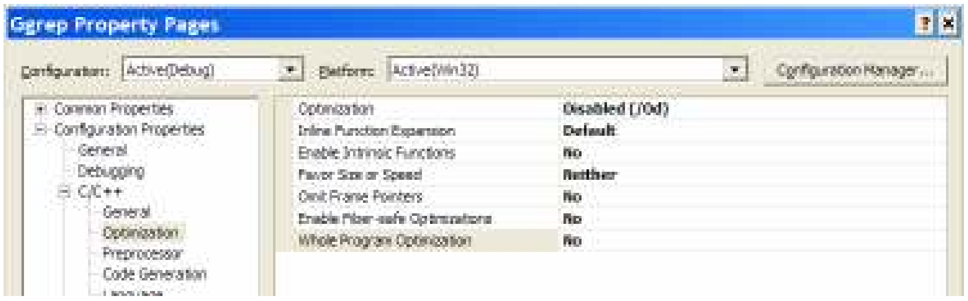


Configuration: Active(Debug) Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	**No**
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	**Both (/RTC1, equiv. to /RTCsu)**
Runtime Library	**Multi-threaded Debug (/MTd)**
Struct Member Alignment	Default
Buffer Security Check	Yes
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

Figure 4.11. Code Generation Tab of Ggrep Optimization On

The optimization off version of Ggrep uses the same source code for each level and property settings of the optimization on version with the exception of changes to the optimization tab (see Figure 4.12). This ensures the same functionality.



Optimization	Disabled (/Od)
Inline Function Expansion	Default
Enable Intrinsic Functions	No
Favor Size or Speed	Neither
Omit Frame Pointers	No
Enable Fiber-safe Optimizations	No
Whole Program Optimization	No

Figure 4.12. Optimization Tab of Ggrep Optimization Off

These two sets of property settings represent the optimization off and optimization on settings for the entire system. This includes Levels 1 through 5 and both benchmark applications.

4.3.2 SciMark2 Development Details

Several of the procedures used for Ggrep are performed for SciMark2 in the exact same manner. These include the wrapping and unwrapping functionality, optimization settings, the use of a location variable and a case statement, the use of static variables, parallelization techniques, and duplicate source code used for each optimization set. Once the slicing of the target function takes place in Level 2, it remains constant for the remaining levels as was the case for Ggrep. At Levels 3-5, changes take place only to the hidden function. Details described in the remainder of this section are limited to function unique details.

Level 1 changes to SciMark2 include setting the number of iterations in each function in kernel.c to constant values and including timing output. The values listed in Table 4.1 allow functions to operate long enough for the system to introduce some randomness through processor usage. These values remain constant for all Levels.

Table 4.1. Iteration Settings for Function Calls

Function	Iterations
FFT	7000
LU	2000
Monte	10000
SOR	250
Sparse	500

Figure 4.13 shows the design of the function calls. The timer starts prior to loop entry. The loop then makes the specified number of calls to the function. After exiting the loop, the timer stops and the length required in seconds is displayed.

```
              ...
              ...
Stopwatch_start(Q);
for (i=0; i<7000; i++)//7000
  {
   FFT_transform(twoN, x);     /* forward transform */
   FFT_inverse(twoN, x);       /* backward transform */
  }
Stopwatch_stop(Q);
printf("FFT took %f seconds\n\n",Stopwatch_read(Q));
              ...
              ...
```

Figure 4.13. FFT function call

Level 2 changes for FFT hides all of the double variables present in the function by placing them into the hidden function H_FFT. A call to H_FFT passes the double *hidden*, the integer *dual*, and the integer *location*. All of the doubles moved to H_FFT are static to retain their values after ending the function calls. Figure 4.14 shows the changes made to FFT_transform_internal. If the value of a variable is needed by the sliced function, it is returned from H_FFT and stored in the local variable *hidden*. The array *data* also remains local to FFT_transform_internal.

```
              ...
              ...
for (bit = 0; bit < logn; bit++, dual *= 2) {
   int a;
   int b;
   hidden=H_FFT(hidden,dual,0) * direction;
   hidden=H_FFT(hidden,dual,1);
   for (a=0, b = 0; b < n; b += 2 * dual) {
     int i = 2*b ;
     int j = 2*(b + dual);
     hidden=H_FFT(data[j],dual,2);
     hidden=H_FFT(data[j+1],dual,3);
     data[j]   = data[i] - H_FFT(data[i],dual,4);
     data[j+1] = data[i+1] - H_FFT(data[i+1],dual,5);
     data[i]  += H_FFT(data[i], dual,4);
     data[i+1]+= H_FFT(data[i+1],dual,5);
    }
   for (a = 1; a < dual; a++) {
     H_FFT(hidden,dual,6);
     for (b = 0; b < n; b += 2 * dual) {
       int i = 2*(b + a);
       int j = 2*(b + a + dual);
       double z1_real = data[j];
       double z1_imag = data[j+1];
       H_FFT(z1_real,dual,7);
       H_FFT(z1_imag,dual,8);
       data[j]   = data[i]   - H_FFT(hidden,dual, 9);
       data[j+1] = data[i+1] - H_FFT(hidden,dual, 10);
       data[i]  += H_FFT(hidden, dual,9);
       data[i+1]+= H_FFT(hidden, dual,10);
      }
   }
 }
              ...
              ...
```

Figure 4.14. Level 2 FFT

Level 2 changes for LU_factor slices out the work being performed and moves it to H_LU. Some of this work includes entire for loops. Figure 4.15 contains the remaining functionality of LU_factor. All of the variables passed to it as parameters are forwarded to H_LU, as well as the current *i*, *j*, and *location*.

```
               ...
               ...
hidden=H_LU(M, N, A, pivot, i, j, 0);
iterations=hidden;
for (j=0; j<iterations; j++)
  {
   hidden=H_LU(M, N, A, pivot, i, j, 1);
   for (i=j+1; i<M; i++)
     hidden=H_LU(M, N, A, pivot, i, j, 2);
   pivot[j]=hidden;
   hidden=H_LU(M, N, A, pivot, i, j, 3);
   if (hidden)
      return 1;  /* factorization failed because of zero pivot */
   hidden=H_LU(M, N, A, pivot, i, j, 4);
   if (j<hidden)
     hidden=H_LU(M, N, A, pivot, i, j, 5);
  }
               ...
               ...
```

Figure 4.15. Level 2 LU

Level 2 changes for MonteCarlo_integrate moves details of calculating the area under the curve and statically stores it in H_Monte. The current value is returned and stored in the local variable *hidden*. The only two calls to H_Monte are presented in Figure 4.16.

```
               ...
               ...
 hidden=H_MonteCarlo(0,0,0);//initialize
 for (count=0; count<Num_samples; count++)
  {
     double x= Random_nextDouble(R);
     double y= Random_nextDouble(R);
     hidden=H_MonteCarlo(x,y,1);
  }
 Random_delete(R);
               ...
               ...
```

Figure 4.16. Level 2 MonteCarlo

Level 2 changes for SOR_execute moves the matrix *G* and its manipulation to H_SOR. The matrix is stored statically in H_SOR. Figure 4.17 presents the only two calls to H_SOR. Although SOR_execute sends its version of the matrix to H_SOR at each call, it is only used when the *location* is 0.

```
                ...
                ...
for (p=0; p<num_iterations; p++)
     {
      for (i=1; i<Mm1; i++)
        {
         hidden=H_SOR(G, omega,0,i);
         for (j=1; j<Nm1; j++)
              hidden=H_SOR(G, omega,1,j);
         }
     }
                ...
                ...
```

Figure 4.17. Level 2 SOR

Level 2 changes for SparseCompRow_matmult moves *sum* and its calculation to H_Sparse. It is only available to SparseCompRow_matmult when it needs to be stored in the array *y*. Figure 4.18 presents the only two calls to H_Sparse.

```
                ...
                ...
for (reps=0; reps<NUM_ITERATIONS; reps++)
     {
         for (r=0; r<M; r++)
         {
             int rowR = row[r];
             int rowRp1 = row[r+1];
                     H_Sparse(val,col,x, 0,0);
             for (i=rowR; i<rowRp1; i++)
                 hidden=H_Sparse(val,col,x, i,1);
             y[r] = hidden;
         }
     }
                ...
                ...
```

Figure 4.18. Level 2 Sparse

Level 3 for FFT parallelizes the switch present in H_FFT from Level 2 with a total of 7 parallel sections. The finalizing of the hidden function, where static variables are assigned the correct temporary value created by the thread, is accomplished with 11 cases. The difference rests in four cases where hidden values are returned, but no additional work is accomplished. Unlike the Level 3 of Ggrep, individual variables are defined prior to the parallel section versus in the *#pragma omp parallel sections* declaration. For example, eight variables are hidden, one for each thread that uses it. Each is initialized prior to entry into the parallel section.

Level 3 for LU parallelizes the switch present in H_LU from Level 2 with a total of 5 parallel sections. The finalizing of the hidden function is accomplished with 6 cases. H_LU contains three parallel sections and if they are executed out of order errors occur. Errors result from attempting to read parts of the matrix which do not yet exist. This problem is resolved with the use of a __try statement. Figure 4.19 shows one of these cases and Figure 4.20 presents two additional cases. Case 2 jumps over the if statement by going to the label *end*. This avoids additional error generation, since *temp_ab* would not contain a value. Case 3 sets the *temp3* flag to 0.

```
#pragma omp parallel sections num_threads(Num_Threads_to_Use)
{
        #pragma omp section
        {//case 0
        temp_minMN= M < N ? M : N;
        }//section
        #pragma omp section
        {//case 1
        temp_jp=j;
        __try{
                temp_t = fabs(A[j][j]);}
        __except(1)
        {;}
        }//section
                ...
                ...
```

Figure 4.19. Case 1 of __try in Level 3 H_LU

Level 3 for Monte parallelizes the switch present in H_Monte from Level 2 with a total of 4 parallel sections. The finalizing of the hidden function is accomplished with 2 cases. Due to the low number of cases present in Monte, two "decoy" sections exist. This maintains the minimum of 4 threads. The "decoy" sections do similar work to calculate the area under the curve. Case 1 in Figure 4.21 is the true thread, while Cases 2 and 3 are the "decoy" threads. Entrance into the if statement is different for all three, as well as the work being done to their local copies of *temp_under_curve*.

```
        ...
        ...
#pragma omp section
{//case 2
__try
{
        temp_ab = fabs(A[i][j]);
}
__except(1)
{goto end;}
if ( temp_ab > t)
 {
   temp2=1;
  temp_jp2 = i;
  temp_t2 = temp_ab;
 }
end:;
}//section
#pragma omp section
{//case 3
__try{
        if( A[jp][j] == 0 )
        temp3=1;}
__except(1){temp3=0;}

}//section
        ...
        ...
```

Figure 4.20. Cases 2 and 3 of __try in Level 3 H_LU

Level 3 for SOR parallelizes the switch present in H_SOR from Level 2 with a total of 4 parallel sections. The finalizing of the hidden function is accomplished with 3 cases. One of the finalizing cases is present for validation purposes only, since the matrix *G* remains hidden in H_SOR. As with Level 3 H_Monte, two "decoy" threads exist. One of the other threads contains a __try statement as well.

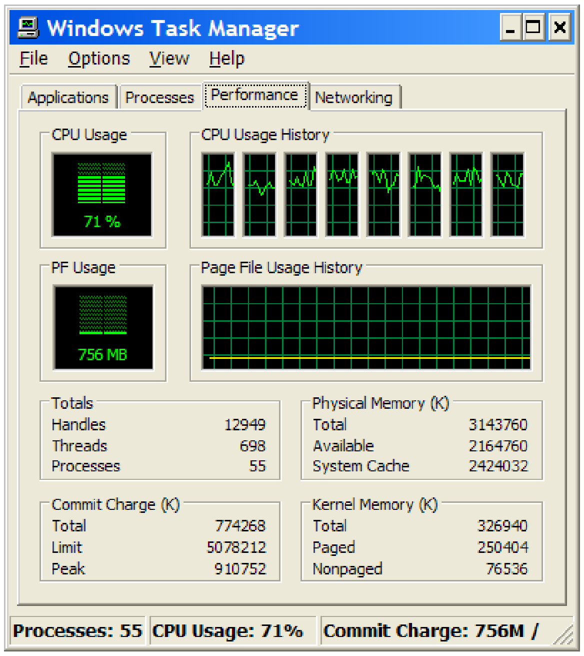
```
            ...
            ...
     #pragma omp section
     {//case 1
            temp_under_curve2=under_curve;
            if ( x*x + y*y <= 1.0)
            {
                   temp1=1;
                   temp_under_curve2 ++;
            }
     }//section
     #pragma omp section
     {//case 2--decoy
            temp_under_curve3=under_curve;
            if ( x*x + y*y == 1.0)
            {
                   temp2=1;
                   temp_under_curve3 --;
            }
     }//section
     #pragma omp section
     {//case 3--decoy
            temp_under_curve4=under_curve;
            if ( x*x + y*y > 1.0)
            {
                   temp3=1;
                   temp_under_curve4=temp_under_curve4+10;
            }
     }//section
            ...
            ...
```

Figure 4.21. Decoy Sections of Level 3 H_Monte

Level 3 for Sparse parallelizes the switch present in H_Sparse from Level 2 with a total of 4 parallel sections. The finalizing of the hidden function is accomplished with 2 cases. Similar to Level 3 H_SOR, two of the threads are "decoy" calculations of *sum* and one thread contains a __try statement.

4.4 System Validation

This section presents the validation procedures used to determine if a function is working properly or not. Changes to code are typically limited to the inclusion of print statements. Each experiment saves output to a text file for validation. The successful completion of the executable is also a measure of validity. Threads are validated to be executing in parallel through print statements distinguishing the thread numbers. Multi-processor validation is accomplish by viewing the Windows Task Manager similar to Figure 4.22



```
Totals                      Physical Memory (K)
Handles          12949      Total           3143760
Threads            698      Available       2164760
Processes           55      System Cache    2424032

Commit Charge (K)           Kernel Memory (K)
Total           774268      Total            326940
Limit          5078212      Paged            250404
Peak            910752      Nonpaged          76536

Processes: 55   CPU Usage: 71%   Commit Charge: 756M /
```

Figure 4.22. Multi-Processor Validation

4.4.1 Ggrep Validation Details

Validation of Ggrep takes place by comparing the text file for all 5 levels to one another for the three separate test expressions used. File comparison using the program KDiff3 [Eib06] determines if the output is correct or not. The files should be exactly the same with the exception of the time required for completion present in the files.

4.4.2 SciMark2 Validation Details

Similar to Ggrep, captured text files are compared using KDiff3 [Eib06]. Unlike Ggrep, each of the five functions requires the inclusion of some print statements to examine certain data points. Figure 4.23 presents the validation for FFT. A static counter is used to determine the cycle the function is currently on. When the counter hits a certain value, the data array prints out a select number of its elements. This same technique is used for LU and Sparse.

```
if (cycle_count==13999){
             printf("FFT validation\n");
             printf("data[0] is %f\n",data[0]);
             printf("data[100] is %f\n",data[100]);
             printf("data[500] is %f\n",data[500]);
             printf("data[1000] is %f\n",data[1000]);
             printf("data[1023] is %f\n",data[1023]);
             cycle_count=0;
      }
      else cycle_count++;
```

Figure 4.23. FFT Validation

The validation of MonteCarlo uses a switch statement to print out the value of the variable *hidden* on certain cycles. *Hidden* contains the value of the variable *under_curve* from H_Monte. Figure 4.24 presents the validation of MonteCarlo.

```
switch(cycle_count){
  case 0:
       printf("MonteCarlo validation\n");
       cycle_count++;
       break;
  case 100:
       printf("Under_curve is %d at cycle %d\n",hidden,cycle_count);
       cycle_count++;
       break;
              ...
              ...
```

Figure 4.24. MonteCarlo Validation

The validation of SOR is accomplished via a call to H_SOR during cycle 249. A call to H_SOR is required, since the current matrix for *G* is present in H_SOR. This call does spin off the parallel sections; however, no work is saved to the static variables, since the switch on the *location* variable leads to only printing out the desired sample points of the matrix *G*.

4.5 Summary

The system under test is comprised of modifications to five functions of SciMark2 and one of Grep. Each system has five levels of obfuscation. The first level is the baseline. The second implements program slicing [ZhG03] with the use of a hidden function. The third adds four parallel threads of execution to the hidden function. The fourth and fifth add an additional four threads each to the hidden function. Validation statements and timing have been added to the functions.

V. Analysis and Results

5.1 Chapter Overview

This chapter presents statistical analysis and results based on the data gathered through the experiments on Ggrep and SciMark2. Analysis is completed in three phases for each system. The first phase looks at the impact of the Levels within the OllyDbg data set with optimization off. The second looks at the impact of the Levels within the OllyDbg data set with optimization on. The last phase compares the two previous sets. Since the executables for the IDAPro and OllyDbg data sets were identical, IDAPro exhibited the same behavior as OllyDbg and therefore IDAPro's analysis with exception of disassembly testing is presented in the Appendix (Tables A.1 through A.23 and Figures A.43 through A.107).

5.2 Ggrep Analysis

5.2.1 Ggrep Analysis, OllyDbg, Optimization-Off

Tables 5.1 through 5.3 present the mean execution time (in seconds), standard deviation, and a 90% confidence interval of Ggrep with Expressions 1 through 3 respectively with OllyDbg and optimization turned off. Five samples are collected at each level.

Table 5.1. Ggrep Expression 1 with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	18.299	0.007	[18.292, 18.305]
2 (hidden function in use)	18.311	0.016	[18.297, 18.326]
3 (hidden function w/ 4 threads)	18.206	0.007	[18.200, 18.213]
4 (hidden function w/ 8 threads)	18.205	0.001	[18.204, 18.205]
5 (hidden function w/ 12 threads)	18.317	0.039	[18.280, 18.354]

Table 5.2. Ggrep Expression 2 with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	5.4066	0.011	[5.3961, 5.4171]
2 (hidden function in use)	5.438	0.038	[5.4018, 5.4742]
3 (hidden function w/ 4 threads)	5.372	0.0067	[5.3656, 5.3784]
4 (hidden function w/ 8 threads)	5.3658	0.0084	[5.3578, 5.3738]
5 (hidden function w/ 12 threads)	5.3658	0.0084	[5.3578, 5.3738]

Table 5.3. Ggrep Expression 3 with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	219.25	0.18	[219.08, 219.43]
2 (hidden function in use)	218.26	0.06	[218.20, 218.32]
3 (hidden function w/ 4 threads)	217.5	0.25	[217.26, 217.73]
4 (hidden function w/ 8 threads)	217.22	0.13	[217.10, 217.34]
5 (hidden function w/ 12 threads)	217.2	0.11	[217.09, 217.30]

It is easy to see that each of the three test expressions result in varying means. This is expected, as the three expressions follow different paths of execution within Ggrep. Although there is a distinct difference in the systems caused by the expression used, it is not as easy to discern if there is a difference among the levels for each separate expression. They appear to be similar, but to determine if a statistically significant difference exists between the levels, the confidence interval of the mean of differences is calculated using the data in Table 5.4 where b_i is the before measurement, a_i is the after measurement, d_i is $b_i - a_i$

Table 5.4. Ggrep Exp1, Mean of Differences Levels 1 and 2

b_i (Level 1)	a_i (Level 2)	$d_i = b_i - a_i$
18.2960	18.3270	-0.0310
18.2960	18.3110	-0.0150
18.3110	18.3270	-0.0160
18.2950	18.2960	-0.0010
18.2950	18.2960	-0.0010

and $(c_1, c_2) = \overline{d} \mp t_{\alpha/2;n-1} \frac{s_d}{\sqrt{n}}$ where $\overline{d}$ is the mean value of d_i which is - 0.0128 seconds, $t_{\alpha/2;n-1}$ is $t_{.05;4}$ which is 2.1318, s_d is the standard deviation of d_i which is 0.0125 seconds, and n is the sample size which is 5. This results in the confidence interval (c_1,c_2) equal to $-0.0128s \mp (2.1318)(\frac{.0125s}{\sqrt{5}})$ which is [-0.0247, -0.0009]. Since the confidence interval does not include zero, there is a statistically significant difference between the execution times of the system set at Level 1 and Level 2 with 90% confidence. Tables 5.5 through 5.7 identify where statistically significant differences are among the Levels for Ggrep executed with Expressions 1 through 3 because the calculated confidence intervals do not include zero. Levels 1 and 2 is the difference between the baseline and the sliced versions of the system. Levels 1 and 3 is the difference between the baseline and the sliced version with 4 parallel threads. Levels 1 and 4 is the difference between the baseline and the sliced version with 8 parallel threads of execution. Levels 1 and 5 is the difference between the baseline and the sliced version with 12 threads of execution. Levels 2 and 3 is the difference between the sliced version and when parallel execution is introduced with only 4 threads. Levels 2 and 4 is the difference between the sliced version and the sliced version with 8 parallel threads. Levels 2 and 5 is the difference between the sliced version and the version with 12 threads. Levels 3 and 4 is the difference between the system with only 4 threads of parallel execution and the version with 8 threads. Levels 3 and 5 is the difference between the system with only 4 threads of parallel execution and the version with 12 threads. Levels 4 and 5 is the difference between the system with 8 threads of parallel execution and 12 threads.

Table 5.5. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 1 with Optimization Off and OllyDbg

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	NO
2 (hidden function in use)	X	X	YES	YES	NO
3 (hidden function w/ 4 threads)	X	X	X	NO	YES
4 (hidden function w/ 8 threads)	X	X	X	X	YES
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table 5.6. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 2 with Optimization Off and OllyDbg

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table 5.7. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 3 with Optimization Off and OllyDbg

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	YES
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

There are differences among 2 of the 3 cases where a hidden function is first introduced in Ggrep. This occurs in the change from Level 1 to Level 2. Statistically significant differences occur among the execution times with all of the expressions from Levels 2 to 3. Once Ggrep had parallel execution introduced in Level 3, there are no statistically significant differences in the execution time by adding four additional threads of execution. In only one case is there no difference detected when going from 4 threads to 12 threads of execution. This was for Expression 2. The remaining expressions do have a statistically significant difference in execution times. In two cases (Expression 2 and 3), there is not a difference when going from 8 threads to 12 threads. In only the case of Expression 1 is there no difference when going from the baseline to a hidden function with 12 threads (Level 1 to Level 5) and from a sliced program with 0 threads to one with 12 threads (Level 2 to Level 5). A change is recognized with Expression 2 and 3 for these two cases.

Looking at the interval plots (Figure 5.1 through 5.3) for each expression, these differences are not obvious in all of the figures. In Figure 5.1, for example, the confidence intervals for Levels 1 and 2 are overlapping. In fact, the lower bound for Level 2 is 18.297 (see Table 5.1). This includes the mean of Level 1 which by visual testing means that there is no difference between the systems at the 90% confidence level. The differences are so small that when taken out to the fourth decimal place the confidence interval of the mean of differences does show a difference. However, rounding all the numbers used in the calculations to only two decimal places shows the interval including 0, which means the systems are not statistically different. The statistically significant difference between Levels 2 and 3 in the system is easy to see visually.

There appears to be a noticeable change in behavior when going from 8 parallel threads to 12 threads. This change is due thread overhead for the particular test expression being used. Although the increase appears large in Figure 5.1, it is important to keep the scale in mind. There is only an increase of .112 seconds. The other two expressions do not have this apparent jump.

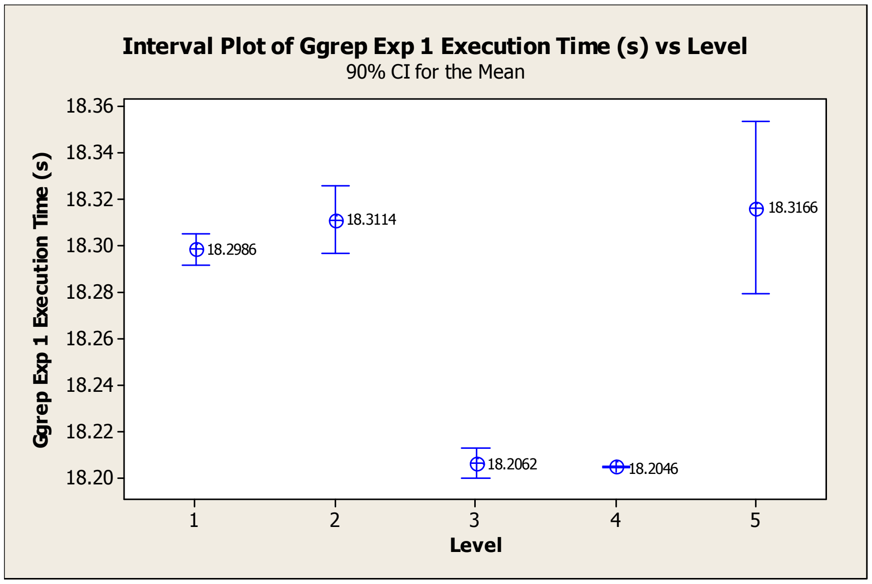


Figure 5.1. Mean Interval Plot of Ggrep Expression 1 with Optimization Off and OllyDbg

Figure 5.2 matches nicely with the statistically significant differences in the execution times of the Ggrep expression 2 in Table 5.7. A simple visual test of the confidence intervals for Levels 1 and 2 shows that they do intersect, meaning there is not a difference between the systems. The same simple test also shows the difference between Levels 2 and 3. There is no difference in the change from Level 3 to Level 4, since the upper bound of the confidence interval for Level 4 is 5.3738 seconds (cf., Table 5.2), thus the interval includes 5.372, the mean of Level 3. The system at Levels 4 and 5 has the same mean and confidence intervals.

The change displayed in Figure 5.3 between Levels 3 and 4 seems small and consistent with the fact that there is not a statistically significant difference between these two levels. The lower bound of Level 3 is 217.26 (cf., Table 5.3). The confidence interval does not include the mean of Level 4, 217.223 seconds. A t-test must be performed to determine if there is a difference, since only the intervals are overlapping. The t-test results in t equal to 2.19. This is larger than its critical value, causing the null hypothesis that the systems are the same to be rejected. This does not correspond with Table 5.6 where the confidence interval of mean of differences identified a difference, since zero was included in the interval. However, if the number of decimal places used in the calculation is decreased to 2 the interval does start at zero. It is easy to see that the system has the same behavior at Levels 4 and 5, where there is a small change. This change, though, is just enough to cause a difference while going from Level 3 to 5. The upper bound for Level 5 is 217.30. This means that neither of the intervals include the other level's mean value, so visually it is non conclusive whether the systems are the same or not and one must rely on the mean of differences calculations or a t-test. The resulting t-value is 2.48, rejecting the null hypothesis that they are the same systems. The confidence interval for the mean of differences also identified that there is a difference between the systems.

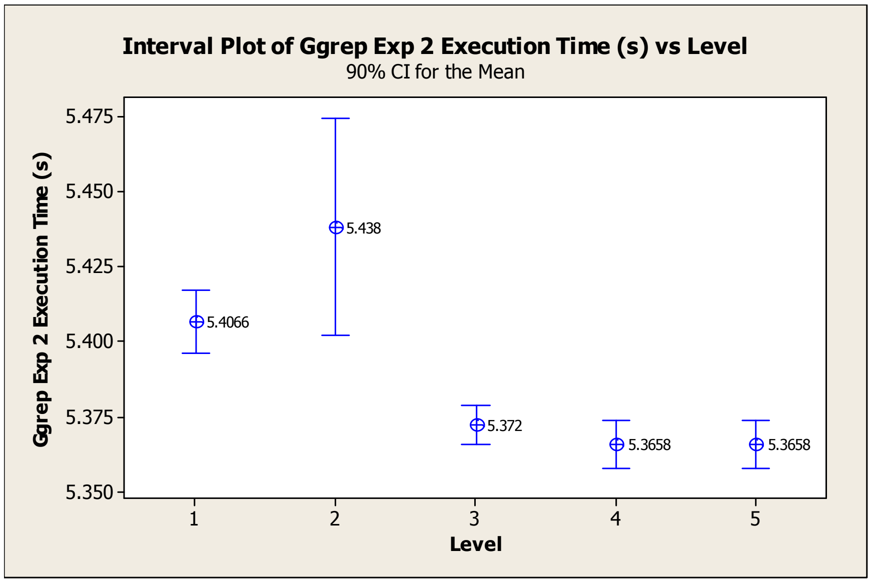


Figure 5.2. Mean Interval Plot of Ggrep Expression 2 with Optimization Off and OllyDbg

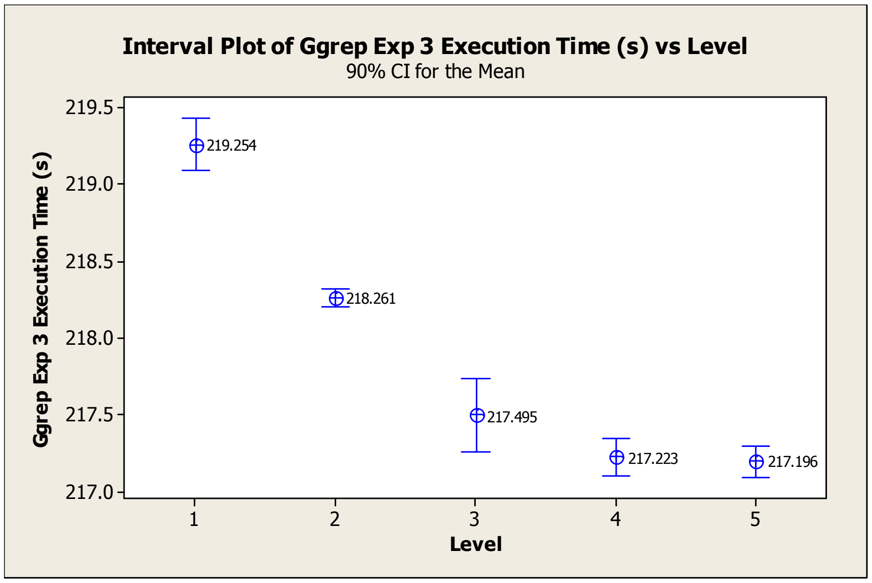


Figure 5.3. Mean Interval Plot of Ggrep Expression 3 with Optimization Off and OllyDbg

Although statistically significant differences are present in the system, practically there is not much of a difference at all. The user of the system would be minimally impacted by differences. The ranges for the levels are .112 seconds for Expression 1, .0722 seconds for Expression 2, and 2.05 seconds for Expression 3. The ideal level for all expressions tested with Ggrep is Level 4. It is possible to introduce 8 threads of execution for the same cost of only 4.

Models were built for all three expressions used with Ggrep, however, the predictive power was extremely weak. Figures A.1 – A.3 in the appendix show the regression equations for Ggrep Expression 1 through 3 respectively. R-Squared values of 0 for Expression 1, .408 for Expression 2 and .828 for Expression 3 are all weak. This causes the models to be unreliable. The 4-in-1 plots for each of the expressions are very similar to the one for Ggrep Expression 1 shown in Figure 5.4. The two additional 4-in-1 plots are Figures A.4 and A.5 of the appendix. There are distinct levels present in each of their Residual versus Fitted Values graphs. This means the standard deviations are correlated to the Levels. The errors are not independent as can be seen in the Residual versus Order Plots. The Histogram of Residuals and the Probability Plots show the errors are not normally distributed.

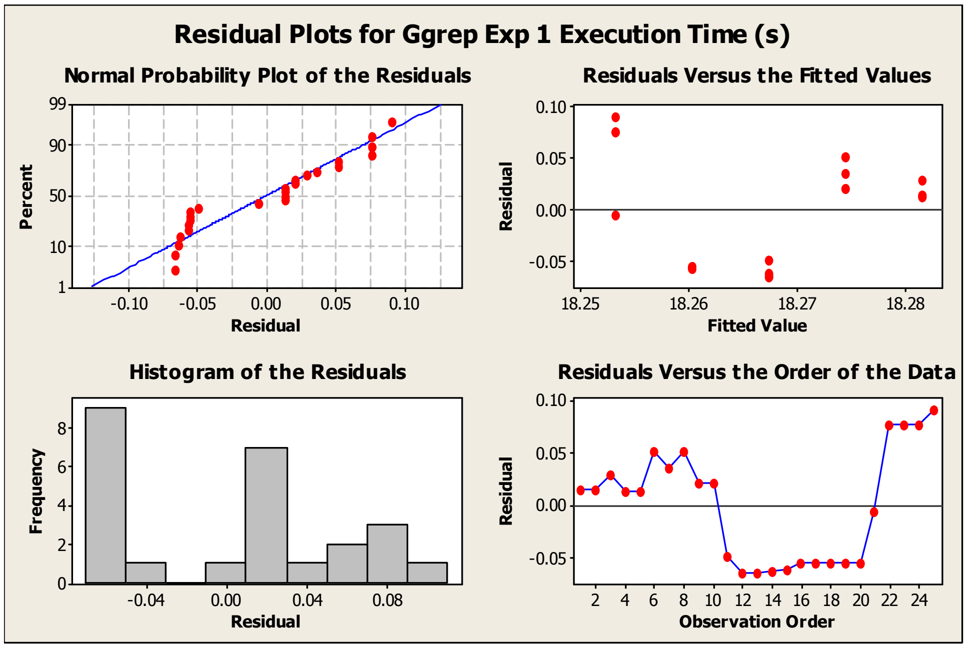


Figure 5.4. 4-in-1 Plot for Ggrep Expression 1 with OllyDbg and Optimization Off, Levels 1-5

5.2.2 Ggrep Analysis, OllyDbg, Optimization-On

Once optimization is turned on for Ggrep, the mean execution times between levels for each expression is even closer as seen in Tables 5.8 - 5.10. Furthermore, it is necessary to validate that the optimization on versions have not removed the parallel threads. Validation is accomplished by inspecting the output and viewing the number threads via the task manager. The insertion of print statements to identify the separate threads of execution at Levels 3-5 could cause a change in the compilers decision on what to optimize. Verification with OllyDbg [Oll05] does show that the correct number of threads are indeed spawned.

Table 5.8. Ggrep Expression 1 with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	16.108	0.076	[16.035, 16.180]
2 (hidden function in use)	16.209	0.093	[16.120, 16.297]
3 (hidden function w/ 4 threads)	16.174	0.051	[16.125, 16.223]
4 (hidden function w/ 8 threads)	16.159	0.082	[16.081, 16.238]
5 (hidden function w/ 12 threads)	16.15	0.029	[16.122, 16.178]

Table 5.9. Ggrep Expression 2 with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	4.7376	0.0069	[4.7310, 4.7442]
2 (hidden function in use)	4.747	0.0067	[4.7406, 4.7534]
3 (hidden function w/ 4 threads)	4.750	0.000	[4.750, 4.750]
4 (hidden function w/ 8 threads)	4.753	0.0067	[4.7466, 4.7594]
5 (hidden function w/ 12 threads)	4.750	0.000	[4.750, 4.750]

Table 5.10. Ggrep Expression 3 with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	193.4	0.24	[193.17, 193.62]
2 (hidden function in use)	194.44	0.06	[194.38, 194.49]
3 (hidden function w/ 4 threads)	194.69	0.17	[194.54, 194.85]
4 (hidden function w/ 8 threads)	194.62	0.12	[194.51, 194.73]
5 (hidden function w/ 12 threads)	194.63	0.1	[194.53, 194.73]

To determine if there are any statistically significant differences among the execution times, a confidence interval for the mean of differences is calculated between each level as accomplished when optimization was turned off. Tables 5.11-5.13 show where the statistically significant differences for the execution times are present in the systems based on the exclusion of zero from the calculated confidence interval.

Expression 1 results in no statistically significant differences present between any of the levels. Expression 2 results in there only being differences when going from the baseline to every other level. There are not statistically significant differences present in the remaining levels. Expression 3 results in there being differences present when going from all levels to another, except from a level with parallel threads to another level with parallel threads.

Table 5.11. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 1 with Optimization On and OllyDbg

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	NO	NO	NO
2 (hidden function in use)	X	X	NO	NO	NO
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table 5.12. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 2 with Optimization On and OllyDbg

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	NO	NO	NO
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table 5.13. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 3 with Optimization On and OllyDbg

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Looking at the interval plots (Figures 5.5 through 5.7) for each expression, these differences are not obvious to see. The user of the system would find it difficult to notice a difference in performance. It is easy to see there is no difference present in the Levels 2 through 5 of Ggrep with Expression 1 as shown in Figure 5.5. The most difficult case to determine that a statistically significant difference is not present is in the change from Level 1 to Level 2. The confidence intervals overlap, but neither of the others mean execution time is included in the others confidence interval. A t-test is required to determine for certain. It results in a t-value of 1.25, therefore the systems are the same.

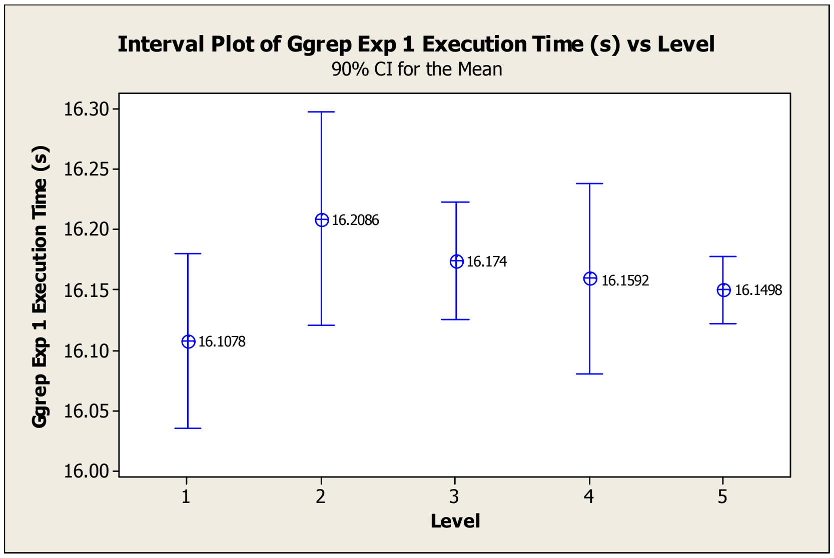


Figure 5.5. Mean Interval Plot of Ggrep Expression 1 with Optimization On and OllyDbg

As with Ggrep Expression 1, the most difficult case to visually distinguish if a difference is present in Ggrep Expression 2 or not is with the change from Level 1 to Level 2. As shown in Figure 5.6, the confidence intervals are overlapping, but the mean of neither level is included by the others confidence interval. This leaves distinguishing the difference to the mean of differences calculation or a t-test, which both show a statistically significant difference. The resulting t-value is -2.18, therefore the null hypothesis does not hold true and there is a difference. Ggrep Expression 2 resulted in the same sample, 4.75 seconds for every repetition of tests at Levels 4 and 5 when Ggrep was executed with optimization on. This causes the confidence intervals for both to be zero, as can be seen in Figure 5.7.

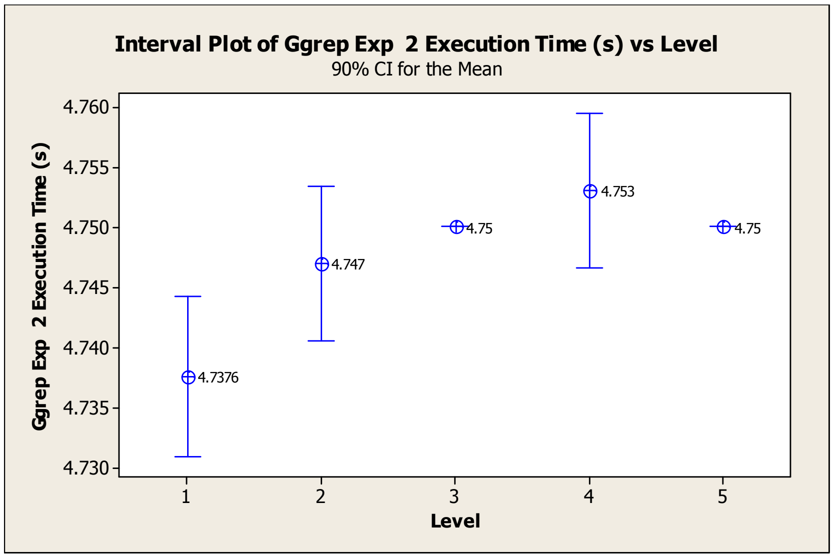


Figure 5.6. Mean Interval Plot of Ggrep Expression 2 with Optimization On and OllyDbg

In the case of Ggrep Expression 3, it is easy to visually distinguish where differences are present in the system (see Figure 5.7). The hardest to see is the change from Level 2 to 3, where the confidence intervals for the execution times come very close, however they do not overlap.

Similar to Ggrep with optimization off, the creation of a model is not of value. The regression models for all three expressions of Ggrep with optimization on and all 5 Levels are extremely weak according to their R-Squared values of 0, .35, and .53 respectively (see Figures A.6 through A.8).

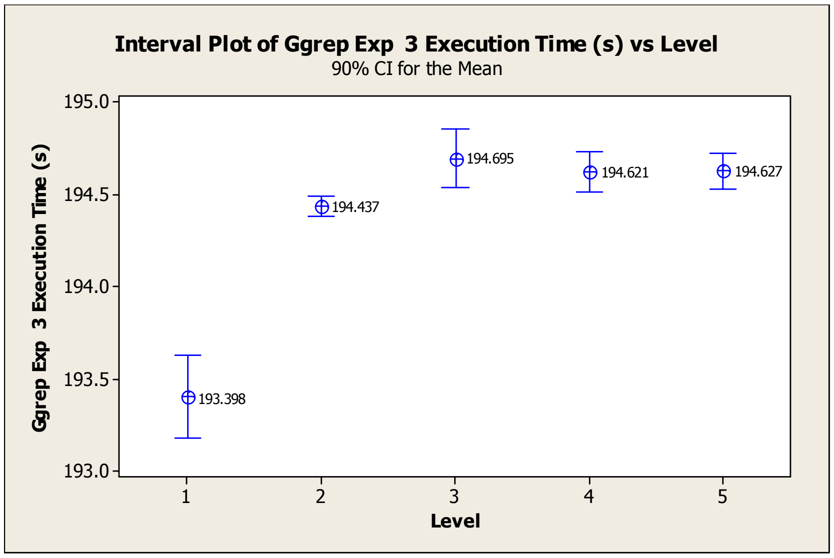


Figure 5.7. Mean Interval Plot of Ggrep Expression 3 with Optimization On and OllyDbg

The 4-in-1 plots have the same properties as Ggrep with optimization on with the exception of Ggrep Expression 2 where the errors have a greater dependence on the level as seen in the Residuals versus Order of the Data plot of Figure 5.8. The 4-in-1 plots of Expressions 1 and 3 are located in the appendix as Figures A.9 and A.10.

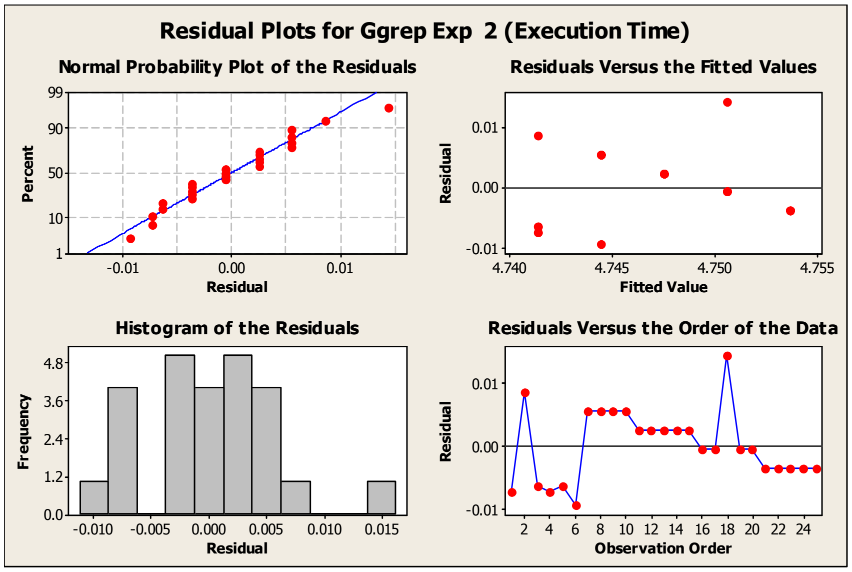


Figure 5.8. 4-in-1 Plot for Ggrep Expression 2 with OllyDbg and Optimization On, Levels 1-5

5.2.3 Ggrep Analysis, OllyDbg, Optimization-Off Versus Optimization-On

Comparing Ggrep with optimization turned off with the version with optimization turned on, the confidence intervals for the mean of differences between the execution times at each level are calculated. There are statistically significant differences between the systems at every level. These differences are easily distinguishable in the combined Interval plots of Ggrep expressions 1 through 3 as shown in Figures 5.9 through 5.11. The highest change in execution time takes place at Level 1 of Expression 3 where an increase of 25.85 seconds is observed (see Table 5.14).

Table 5.14. Change in Execution Time (s) of Ggrep between Optimization Off and Optimization On, OllyDbg and Levels 1-5

Level	Expression 1	Expression 2	Expression 3
1 (baseline)	-2.191	-0.669	-25.85
2 (hidden function in use)	-2.102	-0.691	-23.82
3 (hidden function w/ 4 threads)	-2.032	-0.622	-22.81
4 (hidden function w/ 8 threads)	-2.046	-0.6128	-22.6
5 (hidden function w/ 12 threads)	-2.167	-0.6158	-22.57

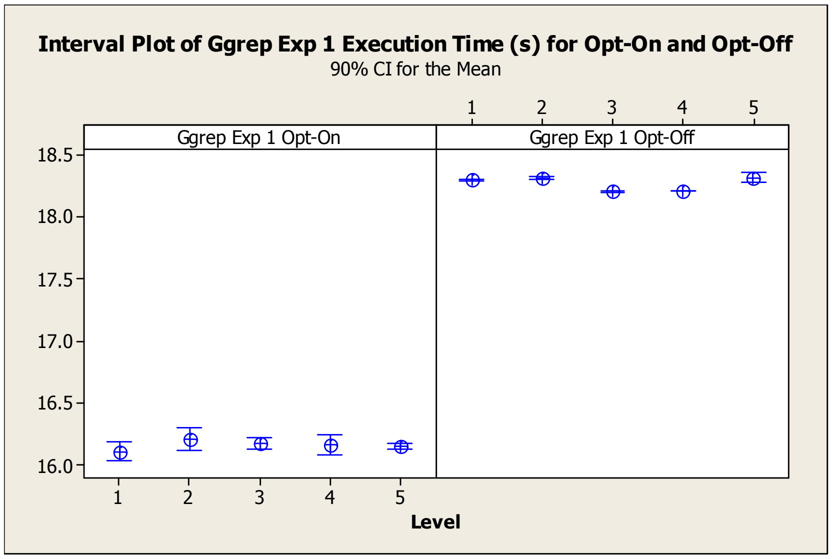


Figure 5.9. Mean Interval Plot of Ggrep Expression 1 with OllyDbg and Optimization On versus Optimization Off

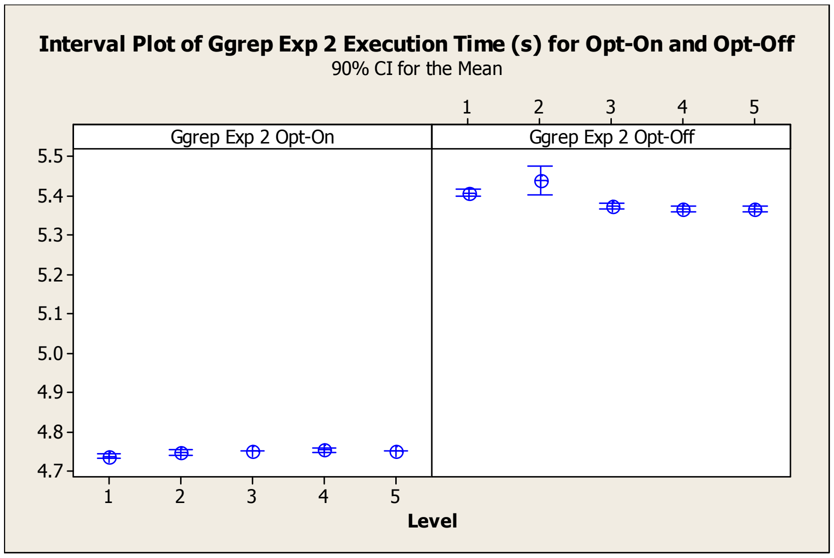


Figure 5.10. Mean Interval Plot of Ggrep Expression 2 with OllyDbg and Optimization On versus Optimization Off

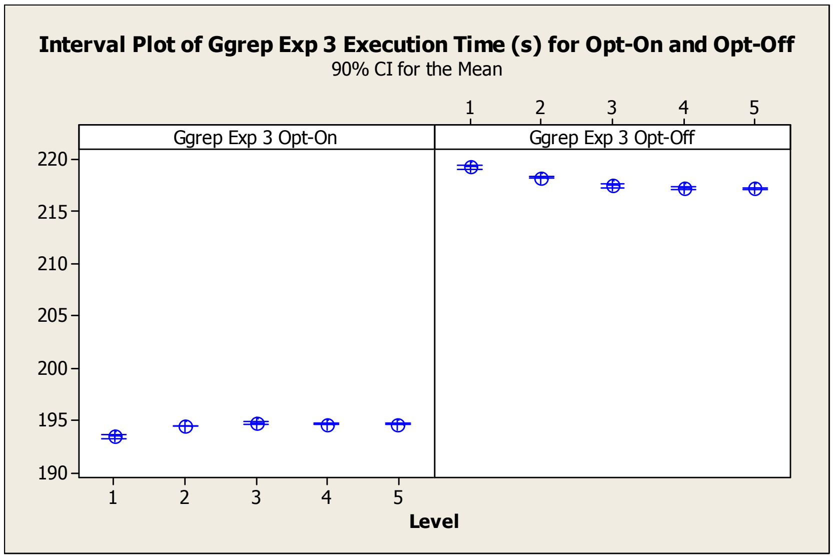


Figure 5.11. Mean Interval Plot of Ggrep Expression 3 with OllyDbg and Optimization On versus Optimization Off

When comparing the file sizes of the executables at each level (see Table 5.11), all of the files decrease in size. Validation has already shown that the threads remain present for Ggrep when optimization is turned on. This decrease in file size, therefore, is not due to the compiler removing the parallel threads.

Table 5.15. Ggrep Executable Size with Optimization Off and On with OllyDbg

Level	Ggrep File Size Opt-Off (bytes)	Ggrep File Size Opt-On (bytes)
1 (baseline)	749568.000	622592.000
2 (hidden function in use)	778240.000	647168.000
3 (hidden function w/ 4 threads)	765952.000	634880.000
4 (hidden function w/ 8 threads)	765952.000	634880.000
5 (hidden function w/ 12 threads)	765952.000	634880.000

5.3 SciMark2 Analysis

5.3.1 SciMark2 Analysis, OllyDbg, Optimization-Off

Tables 5.16 through 5.20 present the mean execution time (in seconds), standard deviation, and a 90% confidence interval of SciMark2 with OllyDbg and Optimization turned off. Five samples are collected at each level. In the baseline version of FFT (Level 1) and the sliced versions with zero threads (Level 2) of LU and SOR, the samples collected within each program were the same causing their standard deviations to be zero. The host system has not introduced enough random delays to cause some variation.

Table 5.16. FFT with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	2.765	0.0000	[2.765, 2.765]
2 (hidden function in use)	9.757	0.0067	[9.7506, 9.7634]
3 (hidden function w/ 4 threads)	15456	1749	[13788, 17123]
4 (hidden function w/ 8 threads)	26159	726	[25466, 26851]
5 (hidden function w/ 12 threads)	38102	698	[37437, 38768]

Table 5.17. LU with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	5.4098	0.0068	[5.4033, 5.4163]
2 (hidden function in use)	5.8280	0.0000	[5.8280, 5.8280]
3 (hidden function w/ 4 threads)	376.12	44.34	[333.85, 418.39]
4 (hidden function w/ 8 threads)	633.3	19.03	[615.16, 651.45]
5 (hidden function w/ 12 threads)	918.49	15.71	[903.51, 933.47]

Table 5.18. Monte with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	4.5968	0.0068	[4.5903, 4.6033]
2 (hidden function in use)	5.8908	0.0004	[5.8904, 5.8912]
3 (hidden function w/ 4 threads)	1748.8	208.9	[1549.6, 1948.0]
4 (hidden function w/ 8 threads)	2951.6	100.8	[2855.4, 3047.7]
5 (hidden function w/ 12 threads)	4324.3	55.9	[4271.0, 4377.6]

Table 5.19. SOR with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	4.7472	0.0129	[4.7349, 4.7595]
2 (hidden function in use)	6.453	0	[6.453, 6.4530]
3 (hidden function w/ 4 threads)	12969	201	[12778, 13161]
4 (hidden function w/ 8 threads)	25460	276	[25197, 25723]
5 (hidden function w/ 12 threads)	36131	203	[35937, 36325]

Table 5.20. Sparse with Optimization Off and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	5.6092	0.0004	[5.6088, 5.6096]
2 (hidden function in use)	13.541	0.083	[13.462, 13.620]
3 (hidden function w/ 4 threads)	25909	3062	[22989, 28829]
4 (hidden function w/ 8 threads)	44086	1468	[42686, 45485]
5 (hidden function w/ 12 threads)	64337	1006	[63378, 65295]

Looking at the interval plot for the execution times of the FFT function of SciMark2, there are differences in the system between Levels 2 through 5 as shown in Figure 5.12. This is also true for the other four functions of SciMark2 and can be seen in the appendix at Figures A.11 through A.14. There is a drastic difference in the required amount of time for the function to complete execution between Level 2 and Level 3. This difference corresponds to when 4 parallel threads are introduced to the system. The means continue to increase with the addition of 4 and 8 threads in Levels 4 and 5.

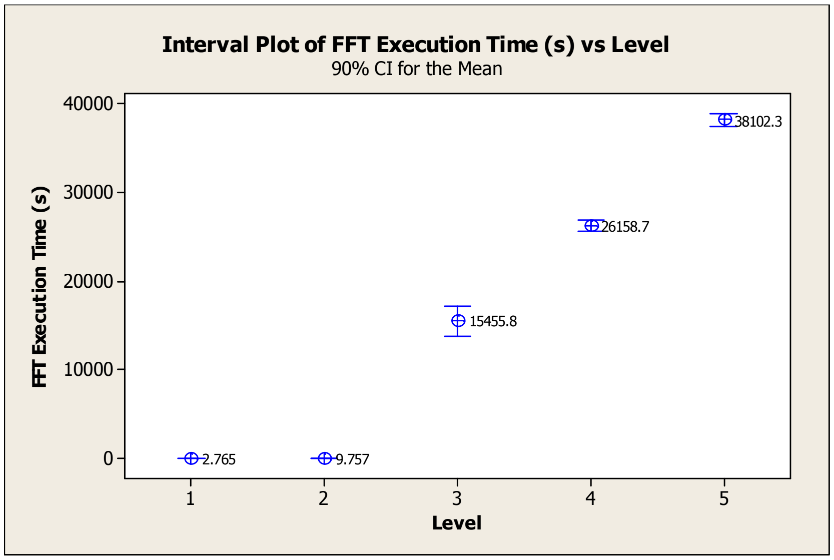


Figure 5.12. Mean Interval Plot of FFT with OllyDbg and Optimization Off

Differences are visible between every level of FFT. To ensure the confidence intervals of Levels 1 and 2 are not intersecting, Figure 5.13 shows only the first two levels for the FFT function of SciMark2. It is very clear that the two levels are different. This is true for the other four functions and can be seen in the appendix at Figures A.15 through A.18. It is not necessary to determine the confidence intervals using the mean of differences of the execution times for comparing any of the levels.

Determining why there is such an extreme change from Level 2 to Level 3 requires examining the mean execution times, the number of calls made to the hidden function, and the number of threads in use. Table 5.21 shows an analysis of the average cost per call per thread. The overall average is 8.2855 x 10^{-6} seconds for each thread associated with a call to the hidden function of the five SciMark2 functions examined. This adds up quickly with millions of calls made to each function.

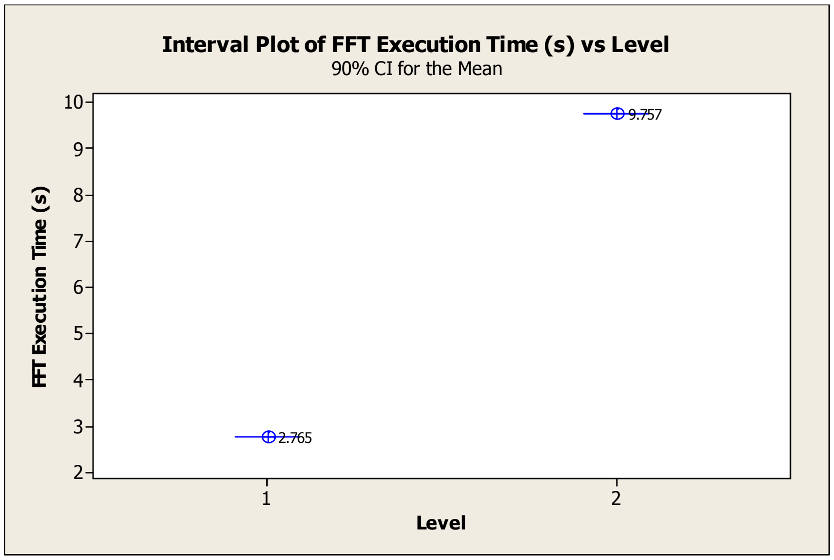


Figure 5.13. Mean Interval Plot of FFT with OllyDbg and Optimization Off Levels 1-2

Table 5.21. Cost Analysis for Threads, Levels 3-5

Level	Num Threads	Function	Mean	Calls to Hidden Function	Time per Call (Mean/Call)	Time per Call/Num Threads
3	4	FFT	15456	444542000	3.47684E-05	8.6921E-06
4	8	FFT	26159	444542000	5.88448E-05	7.3556E-06
5	12	FFT	38102	444542000	8.57107E-05	7.1426E-06
3	4	LU	376.12	10700000	3.51514E-05	8.7879E-06
4	8	LU	633.3	10700000	5.91869E-05	7.3984E-06
5	12	LU	918.49	10700000	8.58402E-05	7.1533E-06
3	4	Monte	1748.8	50005000	3.49725E-05	8.7431E-06
4	8	Monte	2951.6	50005000	5.90261E-05	7.3783E-06
5	12	Monte	4324.3	50005000	8.64774E-05	7.2064E-06
3	4	SOR	12969	301974751	4.29473E-05	1.0737E-05
4	8	SOR	25460	301974751	8.43117E-05	1.0539E-05
5	12	SOR	36131	301974751	0.000119649	9.9708E-06
3	4	Sparse	25909	748500000	3.46146E-05	8.6536E-06
4	8	Sparse	44086	748500000	5.88991E-05	7.3624E-06
5	12	Sparse	64337	748500000	8.59546E-05	7.1629E-06

Attempting to create a model for the system for FFT of SciMark2, an R-squared value of .944 is shown in Figure 5.14. However, this alone is not enough to have a good model. One of the assumptions for a valid model is that the errors are normally distributed. It can be seen in the Histogram of Residuals and the Normal Probability plots of the 4-in-1 Plot for FFT in Figure 5.15 that the errors are not normally distributed. Also, the Residual versus the Order of the Data shows dependence at Levels 1 and 2. The Residuals versus the fitted values shows grouping among the levels in the standard deviation. These properties also hold for the LU, Monte, SOR, and Sparse functions of SciMark2. Their corresponding regression models and 4-in-1 plots are located in the appendix at Figures A.19-A.26.

```
The regression equation is
FFT Execution Time (s) = - 14759 + 10235 Level

Predictor      Coef  SE Coef      T      P
Constant     -14759     1685  -8.76  0.000
Level       10234.8    507.9  20.15  0.000

S = 3591.64   R-Sq = 94.6%   R-Sq(adj) = 94.4%

Analysis of Variance

Source          DF          SS          MS       F      P
Regression       1  5237554341  5237554341  406.01  0.000
Residual Error  23   296697964    12899911
Total           24  5534252305
```

Figure 5.14. Regression Model of FFT with OllyDbg and Optimization Off, Levels 1-5

5.3.2 SciMark2 Analysis, OllyDbg, Optimization-On

Tables 5.22 through 5.26 contain the mean execution time (in seconds), standard deviation, and a 90% confidence interval of SciMark2 with OllyDbg and Optimization turned on. Five samples are collected at each level. Levels 3 through 5 for the Monte function of SciMark2 fail to execute. This is the same exact code used with the optimization off data sets, which did execute and validate successfully, but in this case the compiler has removed something necessary for the function to operate correctly. Using OllyDbg to validate that the parallel threads remain for SciMark2 when optimization is turned on reveals that the compiler has completely removed them from Levels 3 through 5.

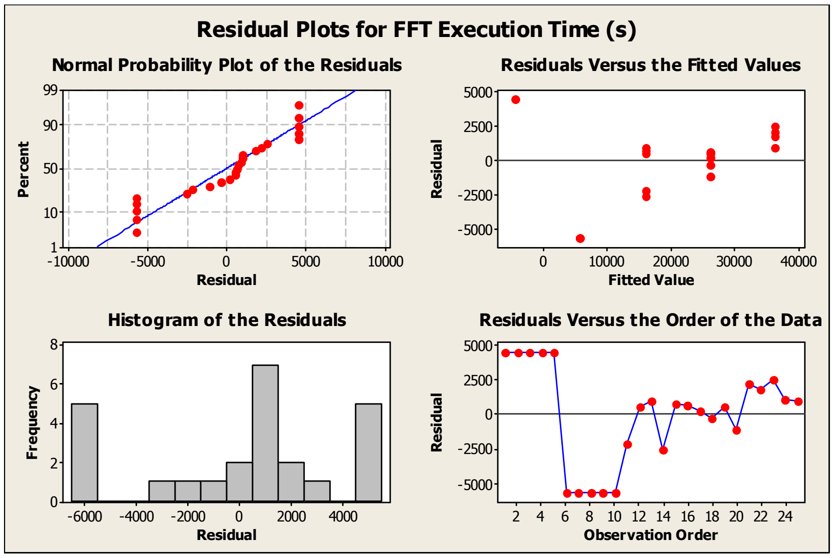


Figure 5.15. 4-in-1 Plot of FFT with OllyDbg and Optimization Off, Levels 1-5

Table 5.22. FFT with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	3.9242	0.0072	[3.9174, 3.9310]
2 (hidden function in use)	10.081	0.06	[10.023, 10.138]
3 (hidden function w/ 4 threads)	69.472	0.026	[69.447, 69.496]
4 (hidden function w/ 8 threads)	69.457	0.078	[69.383, 69.531]
5 (hidden function w/ 12 threads)	69.5	0.08	[69.424, 69.576]

Table 5.23. LU with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	5.0032	0.0072	[4.9964, 5.0100]
2 (hidden function in use)	5.4252	0.0072	[5.4184, 5.4320]
3 (hidden function w/ 4 threads)	10.344	0.058	[10.288, 10.399]
4 (hidden function w/ 8 threads)	10.316	0.023	[10.293, 10.338]
5 (hidden function w/ 12 threads)	10.3	0.017	[10.284, 10.316]

Table 5.24. Monte with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	4.5756	0.0071	[4.5689, 4.5823]
2 (hidden function in use)	4.8092	0.0068	[4.8027, 4.8157]
3 (hidden function w/ 4 threads)	N/A	N/A	N/A
4 (hidden function w/ 8 threads)	N/A	N/A	N/A
5 (hidden function w/ 12 threads)	N/A	N/A	N/A

Table 5.25. SOR with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	4.7438	0.0085	[4.7357, 4.7519]
2 (hidden function in use)	7.2314	0.0069	[7.2248, 7.2380]
3 (hidden function w/ 4 threads)	20.628	1	[19.674, 21.581]
4 (hidden function w/ 8 threads)	19.818	1.116	[18.754, 20.882]
5 (hidden function w/ 12 threads)	20.29	2.097	[18.291, 22.289]

Table 5.26. Sparse with Optimization On and OllyDbg, Levels 1-5

Level	Mean	St-Dev	90% Confidence Interval
1 (baseline)	5.6058	0.0134	[5.5930, 5.6186]
2 (hidden function in use)	13.291	0.008	[13.283, 13.299]
3 (hidden function w/ 4 threads)	47.897	0.039	[47.860, 47.935]
4 (hidden function w/ 8 threads)	47.903	0.064	[47.842, 47.964]
5 (hidden function w/ 12 threads)	47.882	0.021	[47.861, 47.902]

Although the parallel threads have been removed, the interval plots of each of the programs are examined for the sake of completeness. In Figure 5.16 there are evident differences between Levels 1-2 and Levels 2-3 for FFT. This holds true for the other four functions of SciMark2 and can be seen the appendix at Figures A.27 through A.30. It is difficult, however, to determine if differences are present between Levels 3 through 5 in all of the functions except for Monte which does not have Levels 3 through 5 present. Confidence intervals of the mean of differences in the execution times among the three levels are calculated for each of the four functions. These confidence intervals reveal a single statistically significant difference between the execution times at Levels 4 and 5 of FFT.

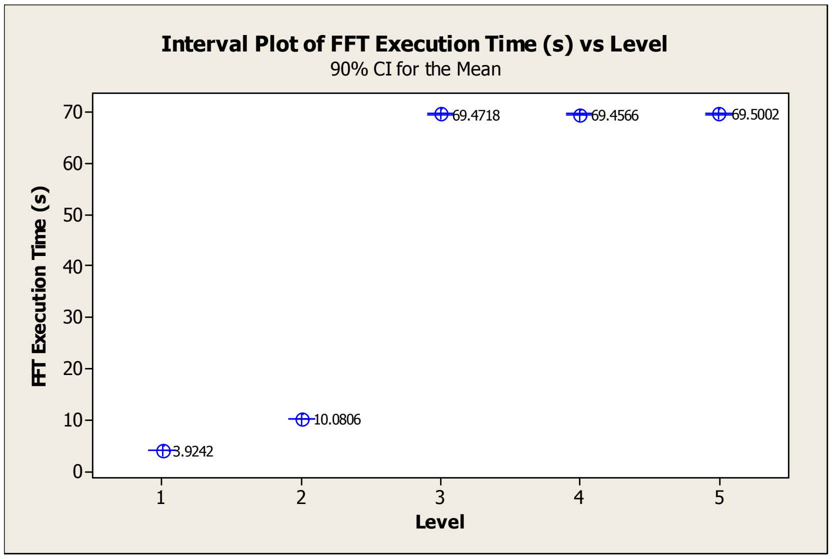


Figure 5.16. Mean Interval Plot of FFT with OllyDbg and Optimization On Levels 1-5

Models were also attempted for FFT. However, as with SciMark2 with optimization off the models for SciMark2 with optimization on are not valid because of the properties of the errors present. Figure 5.17 shows the regression model with an R-squared value of .762 for FFT with optimization on.

```
The regression equation is
FFT (Execution Time) = - 12.7 + 19.1 Level

Predictor      Coef  SE Coef      T      P
Constant    -12.672    7.162  -1.77  0.090
Level        19.053    2.159   8.82  0.000

S = 15.2689   R-Sq = 77.2%   R-Sq(adj) = 76.2%

Analysis of Variance

Source          DF     SS     MS      F      P
Regression       1  18150  18150  77.85  0.000
Residual Error  23   5362    233
Total           24  23513
```

Figure 5.17. Regression Model of FFT with OllyDbg and Optimization On, Levels 1-5

The 4-in-1 plot for FFT of SciMark2 with optimization on is contained in Figure 5.18. The Residuals versus the Order of Data plot for FFT shows that the errors are dependent on the level. Its Residual versus Fitted Values plot clearly distinguishes the five separate levels. The Normal Probability and the Histogram of the Residuals plots of do not show a normal distribution for the errors. These do not support a valid model.

These same properties are present in the remaining four functions of SciMark2 with optimization on. The regression models and the associated 4-in-1 plots are Figures A.31 through A.38 of the appendix.

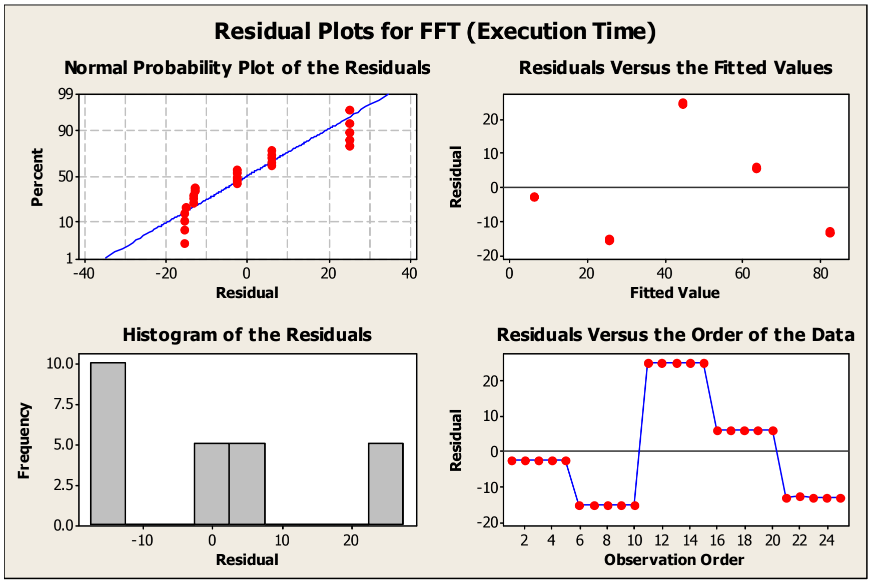


Figure 5.18. 4-in-1 Plot of FFT with OllyDbg and Optimization On, Levels 1-5

5.3.3 SciMark2 Analysis, OllyDbg, Optimization-Off Versus Optimization-On

Comparing the two different versions of SciMark2 is limited to just Levels 1 and 2 since the compiler removed the parallel threads of Levels 3 through 5 with optimization on. Figure 5.19 shows the confidence intervals for Monte with optimization on and off.

The most difficult level to determine whether a difference is present in the execution time is the baseline version, Level 1. This was not true for FFT and LU where the differences between the levels are easily distinguishable (see Figures A.39 and A.40 of the appendix). However, it is true for SOR and Sparse (see Figures A.41 and A.42). A confidence interval of the mean of differences is calculated to determine if there are statistically significant differences between the Level 1 versions of Monte, SOR, and Sparse. The confidence intervals reveal that the SOR and Sparse functions are statistically equivalent with optimization on and off. There is a difference in the Monte function.

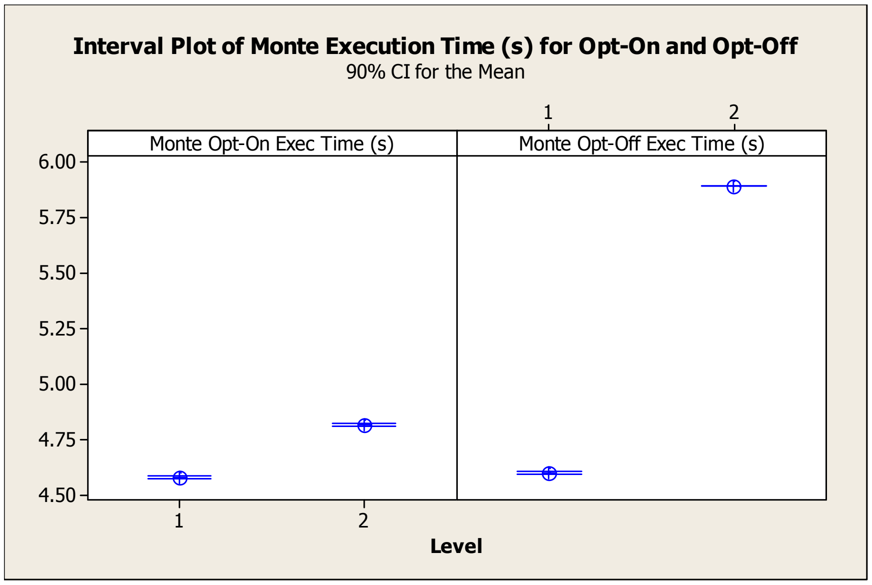


Figure 5.19. Mean Interval Plot of Monte with Optimization On and Off with OllyDbg, Levels 1-2

Comparing the file sizes for the version with optimization off and on shows a decrease at every level. This is the same as Ggrep. Table 5.27 shows the executable file sizes in bytes.

Table 5.27. File Sizes for SciMark2 with Optimization On and Off with OllyDbg

Level	SciMark2 File Size Opt-Off (bytes)	SciMark2 File Size Opt-On (bytes)
1 (baseline)	53248	24576
2 (hidden function in use)	65536	36352
3 (hidden function w/ 4 threads)	69632	38400
4 (hidden function w/ 8 threads)	69632	38400
5 (hidden function w/ 12 threads)	69632	38400

5.4 Disassembly with OllyDbg and IDAPro

Relying on the experimental assumption that each of the hidden functions execute in a secure area prevents debuggers from disassembling the hidden functions themselves. This would essentially be equivalent to removing the hidden functions entirely from the executable and attempting to disassemble them with a debugger. The lack of the hidden function would cause application failure.

Setting this assumption aside and analyzing the system at the levels containing parallel threads is extremely worthwhile. Table 5.28 summarizes when disassembly fails in the system. Recall that SciMark2 with optimization on has no parallel threads due to compiler actions. Therefore, it is not present in the table.

Disassembly of Levels 3 through 5 for Ggrep with optimization on and off is possible with both OllyDbg and IDAPro debuggers. Both debuggers are capable of disassembling and executing Ggrep with all three expressions without error. However, when break points are set in an attempt to determine the functionality of the separate threads, both debuggers have problems stepping through the additional threads.

Table 5.28. Disassembly Results, Levels 3 through 5

	Is OllyDbg capable of Executing?	Is OllyDbg capable of Executing with break point set?	Is IDAPro capable of Executing?	Is IDAPro capable of Executing with break point set?
Ggrep Exp1 Optimization Off	Yes	No	Yes	No
Ggrep Exp1 Optimization On	Yes	No	Yes	No
Ggrep Exp2 Optimization Off	Yes	No	Yes	No
Ggrep Exp2 Optimization On	Yes	No	Yes	No
Ggrep Exp3 Optimization Off	Yes	No	Yes	No
Ggrep Exp3 Optimization On	Yes	No	Yes	No
SciMark2 Optimization Off	Yes	No	Yes	No

With OllyDbg, problems occur after executing the target thread until it completes and enters a sleep state. It remains active with the other threads in a paused state. At this point it is necessary to pause the target thread, since it is in a locked state. This leaves all the threads in a paused state. After giving control back to the main thread and continuing to attempt debugging, an access violation is encounter when returning back to one of the additional threads. This behavior is not present during normal execution.

With IDAPro, a similar problem occurs leaving the program with warnings of memory write problems. The break point is set at address 0040390F in the case of both debuggers.

Disassembly of Levels 3 through 5 for SciMark2 with optimization off using OllyDbg leads to access violations. This occurs with and without a breakpoint set at 0040181A. When using IDAPro, SciMark2 executes correctly without a break point set.

When the break point is set, the same problems occur that were present when Ggrep is attempted to be executed.

These problems combined with the presence of multiple parallel threads make it difficult for an attacker to track down the correct flow of execution. Stepping through the execution is normal until the threads are started. This can be a deterrent, as the resources required to determine the functionality increase with the number of threads.

5.5 Summary

This chapter analyzes the Ggrep and SciMark2 systems. Ggrep's performance with optimization off and optimization on are similar. Although there are statistically significant differences at every level, a user would be minimally impacted by the difference in execution times. SciMark2 is statistically different at every level except for two functions (SOR, Sparse) at the baseline level. There is a drastic cost associated with implementing parallel threads with SciMark2 when optimization is off. Execution of each thread averages 8.2855 x 10^{-6} seconds. This hinders performance when the number of function calls in SciMark2 are in the millions. The compiler removes the parallel threads from SciMark2 when optimization is turned on. Both debuggers experience problems when disassembling both Ggrep and SciMark2 when parallel threads are present.

VI. Conclusions and Recommendations

6.1 Chapter Overview

This chapter presents the conclusions and the contributions of this research. It concludes with recommendations for further areas of study which include parallel thread execution.

6.2 Conclusions of Research

This research proves that the inclusion of parallel threads with the concept of program slicing [ZhG03] with a hidden function is a viable means of software security. The cost of parallel threads should, however, be considered if calls to the hidden function become extremely large as in the case of SciMark2. If the number of calls remain relatively few, there will be no statistically significant difference between the baseline of an application and the inclusion of twelve threads of parallel execution. Ggrep with Expression 1 and optimization turned on demonstrates this.

The use of parallel threads alone makes disassembly with a debugger more difficult. In some cases, the debugger may experience problems as when break points are introduced in both Ggrep and SciMark2.

Compiler optimizations can remove parallel threads during the compilation process as seen in the SciMark2 system. However, this is not always the case since the threads remained when optimization is turned on for Ggrep.

In both Ggrep and SciMark2, increasing the number of threads present in an application did not increase the executable size stored on disk. However, while the increase from four to eight and eight to twelve threads has no impact on file size, there is an increase when introducing threading to single threaded applications.

6.3 Research Contributions

Software security should take advantage of multi-processor technology supported by most computers. This research uses multi-threading with OpenMP and combines two previously proven concepts. The first being the use of hiding program slices for security [ZhG03], while the second is using parallel threads as a means of obfuscation [CTL97]. This research executes parallel threads from within a hidden function and proves it as a viable option. It is possible to easily introduce multiple false paths of execution which can perform similar or non-related work to mislead an attacker from the true functionality of the program.

There is a limitation associated with the use of threads for security. The added execution time associated with each thread adds up quickly if a large number of calls to the hidden function are made.

Even so, this technique for software security can be directly applied to applications developed for and by the Air Force.

6.4 Recommendations for Future Research

Since the demand for speed and performance are driving most computers to include multi-processors, using parallel threads for security warrants further examination. Some additional research topics are proposed below.

Implement this concept on a secure device. Instead of simulating the secure device as in this research, implementation should be done on a real device. Although it may be possible to execute an entire application in a secure area, resource restrictions will likely not permit this.

Implement a security thread to detect any modifications. This thread could run continually to check for modifications to the application. Upon detection, it could stop the application from executing. This concept could also be implemented to detect the presence of a debugger on a system.

Implement parallel threads which continuously check one another. If a thread realizes another thread has stopped, as is the case when attempting to disassemble with a debugger, the application could be stopped.

Parallel threads with metamorphic code. The concept of metamorphic code [Dub06] along with parallel threads of execution has strong potential for security.

6.5 Summary

The implementation of security through the use of parallel threads being executed from a secure hidden function has both strengths and weaknesses. The concept is proven possible, but is limited due to the cost associated with the use of threads. Care must be taken to ensure parallel threads are not removed by a compiler if optimization is used.

Appendix

```
The regression equation is
Ggrep Exp 1 Execution Time (s) = 18.3 - 0.00708 Level

Predictor        Coef    SE Coef       T      P
Constant      18.2887     0.0259  706.90  0.000
Level       -0.007080   0.007801   -0.91  0.373

S = 0.0551590   R-Sq = 3.5%   R-Sq(adj) = 0.0%


Analysis of Variance

Source          DF        SS        MS     F      P
Regression       1  0.002506  0.002506  0.82  0.373
Residual Error  23  0.069978  0.003043
Total           24  0.072484
```

Figure A.1. Regression Model for Ggrep Expression 1 with OllyDbg and Optimization Off, Levels 1-5

```
The regression equation is
Ggrep Exp  2 (Execution Time) = 5.44 - 0.0154 Level

Predictor        Coef    SE Coef       T      P
Constant      5.43578    0.01219  445.90  0.000
Level       -0.015380   0.003676   -4.18  0.000

S = 0.0259904   R-Sq = 43.2%   R-Sq(adj) = 40.8%

Analysis of Variance

Source          DF        SS        MS      F      P
Regression       1  0.011827  0.011827  17.51  0.000
Residual Error  23  0.015537  0.000676
Total           24  0.027364

Unusual Observations

             Ggrep Exp
                     2
            (Execution
Obs  Level       Time)      Fit    SE Fit  Residual  St Resid
  8   2.00     5.50000  5.40502  0.00637   0.09498      3.77R

R denotes an observation with a large standardized residual.
```

Figure A.2. Regression Model for Ggrep Expression 2 with OllyDbg and Optimization Off, Levels 1-5

```
The regression equation is
Ggrep Exp  3 (Execution Time) = 219 - 0.515 Level

Predictor       Coef  SE Coef        T      P
Constant     219.432    0.158  1386.52  0.000
Level       -0.51544  0.04772   -10.80  0.000

S = 0.337415   R-Sq = 83.5%   R-Sq(adj) = 82.8%

Analysis of Variance

Source          DF      SS      MS       F      P
Regression       1  13.284  13.284  116.68  0.000
Residual Error  23   2.619   0.114
Total           24  15.902

Unusual Observations

             Ggrep Exp
                     3
            (Execution
Obs  Level       Time)      Fit  SE Fit  Residual  St Resid
 15   3.00     217.172  217.886   0.067    -0.714    -2.16R

R denotes an observation with a large standardized residual.
```

Figure A.3. Regression Model for Ggrep Expression 3 with OllyDbg and Optimization Off, Levels 1-5

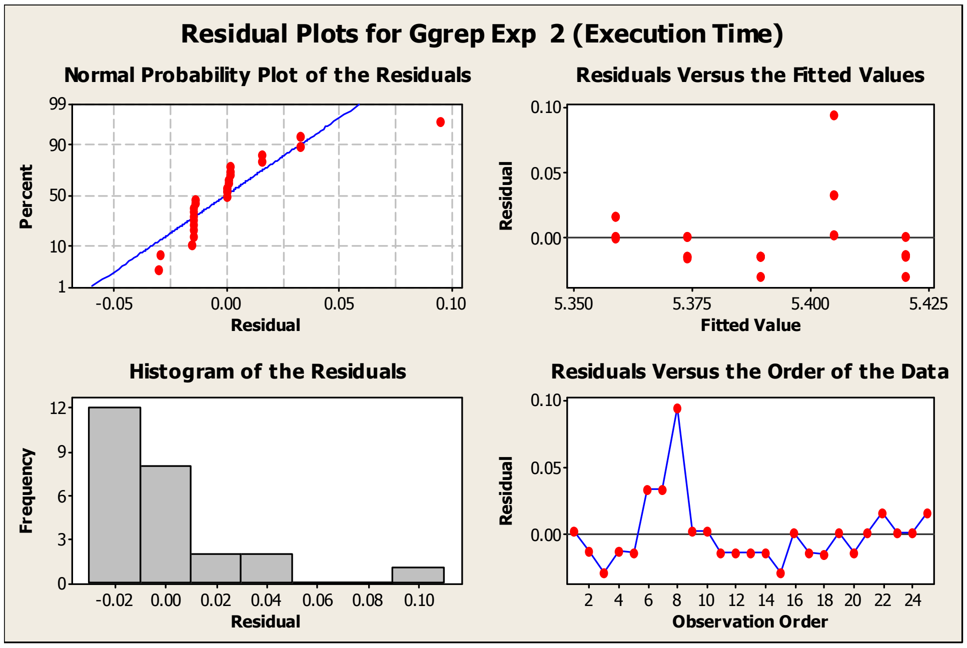


Figure A.4. 4-in-1 Plot for Ggrep Expression 2 with OllyDbg and Optimization Off, Levels 1-5

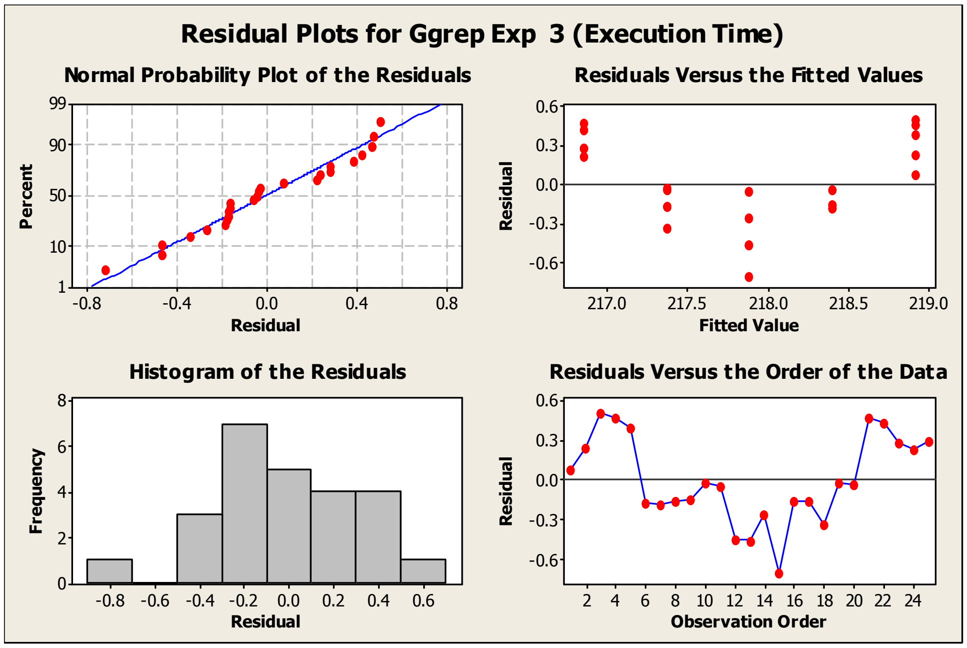


Figure A.5. 4-in-1 Plot for Ggrep Expression 3 with OllyDbg and Optimization Off, Levels 1-5

```
The regression equation is
Ggrep Exp 1 (Execution Time) = 16.1 + 0.0035 Level

Predictor     Coef   SE Coef       T      P
Constant   16.1495    0.0346  467.38  0.000
Level      0.00346   0.01042    0.33  0.743

S = 0.0736685   R-Sq = 0.5%   R-Sq(adj) = 0.0%

Analysis of Variance

Source          DF        SS        MS     F      P
Regression       1  0.000599  0.000599  0.11  0.743
Residual Error  23  0.124822  0.005427
Total           24  0.125421

Unusual Observations

             Ggrep Exp
                     1
            (Execution
Obs  Level       Time)      Fit  SE Fit  Residual  St Resid
  9   2.00     16.3680  16.1564  0.0180    0.2116     2.96R

R denotes an observation with a large standardized residual.
```

Figure A.6. Regression Model for Ggrep Expression 1 with OllyDbg and Optimization On, Levels 1-5

```
The regression equation is
Ggrep Exp  2 (Execution Time) = 4.74 + 0.00308 Level

Predictor        Coef     SE Coef        T      P
Constant      4.73828     0.00274  1730.36  0.000
Level       0.0030800   0.0008256     3.73  0.001

S = 0.00583811   R-Sq = 37.7%   R-Sq(adj) = 35.0%

Analysis of Variance

Source          DF          SS          MS      F      P
Regression       1  0.00047432  0.00047432  13.92  0.001
Residual Error  23  0.00078392  0.00003408
Total           24  0.00125824

Unusual Observations

             Ggrep Exp
                     2
            (Execution
Obs  Level       Time)      Fit   SE Fit  Residual  St Resid
 18   4.00     4.76500  4.75060  0.00143   0.01440      2.54R

R denotes an observation with a large standardized residual.
```

Figure A.7. Regression Model for Ggrep Expression 2 with OllyDbg and Optimization On, Levels 1-5

```
The regression equation is
Ggrep Exp  3 (Execution Time) = 194 + 0.264 Level

Predictor      Coef  SE Coef        T      P
Constant    193.563    0.165  1169.77  0.000
Level       0.26416  0.04989     5.29  0.000

S = 0.352786   R-Sq = 54.9%   R-Sq(adj) = 53.0%

Analysis of Variance

Source          DF      SS      MS      F      P
Regression       1  3.4890  3.4890  28.03  0.000
Residual Error  23  2.8625  0.1245
Total           24  6.3516

Unusual Observations

             Ggrep Exp
                     3
            (Execution
Obs  Level       Time)      Fit  SE Fit  Residual  St Resid
  4   1.00     193.156  193.827   0.122    -0.671    -2.03R

R denotes an observation with a large standardized residual.
```

Figure A.8. Regression Model for Ggrep Expression 3 with OllyDbg and Optimization On, Levels 1-5

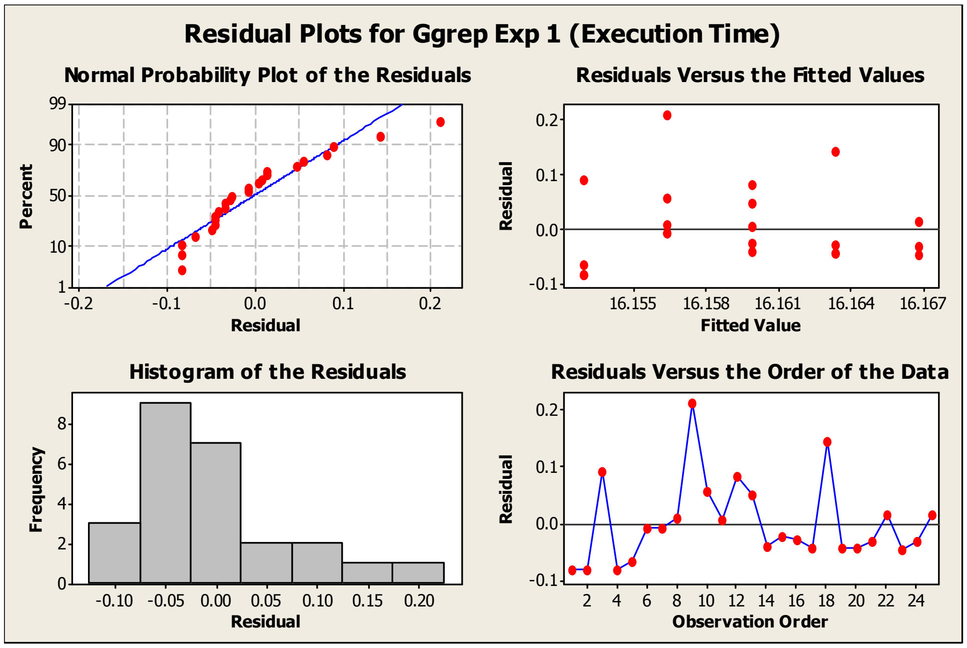


Figure A.9. 4-in-1 Plot for Ggrep Expression 1 with OllyDbg and Optimization On, Levels 1-5

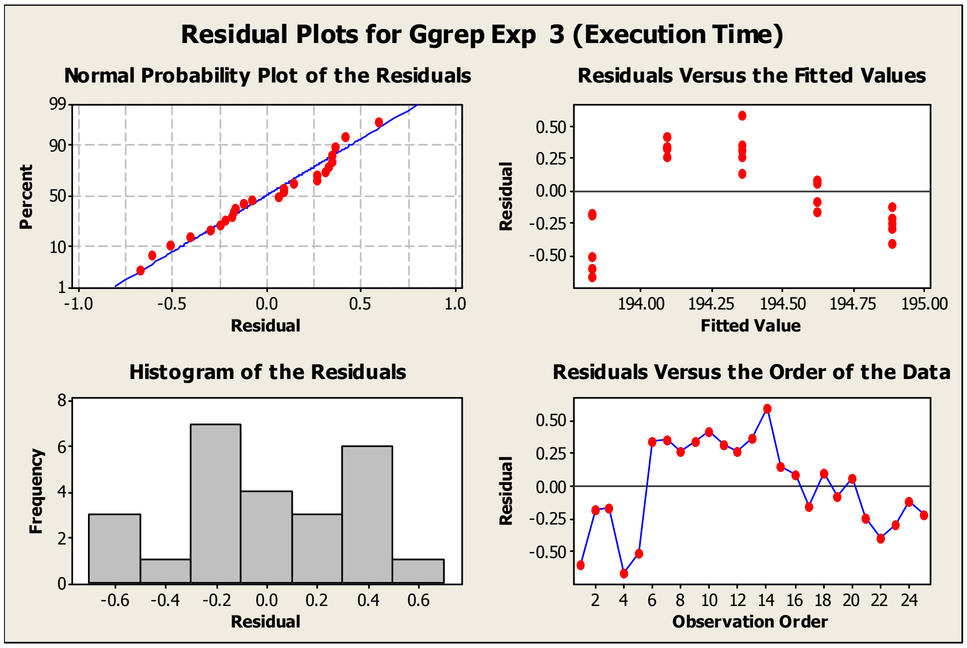


Figure A.10. 4-in-1 Plot for Ggrep Expression 3 with OllyDbg and Optimization On, Levels 1-5

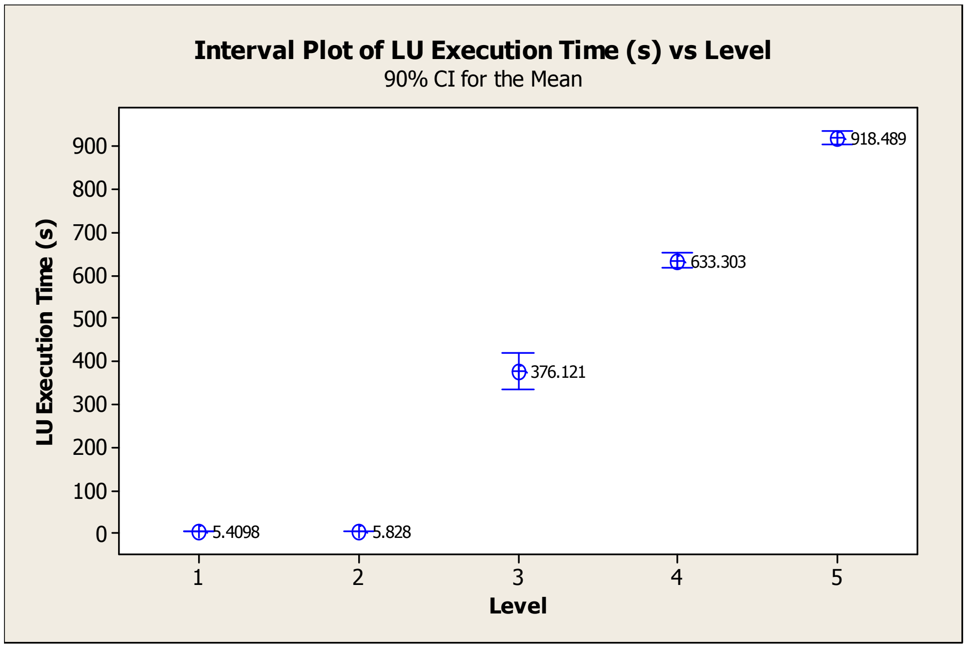


Figure A.11. Mean Interval Plot of LU with OllyDbg and Optimization Off

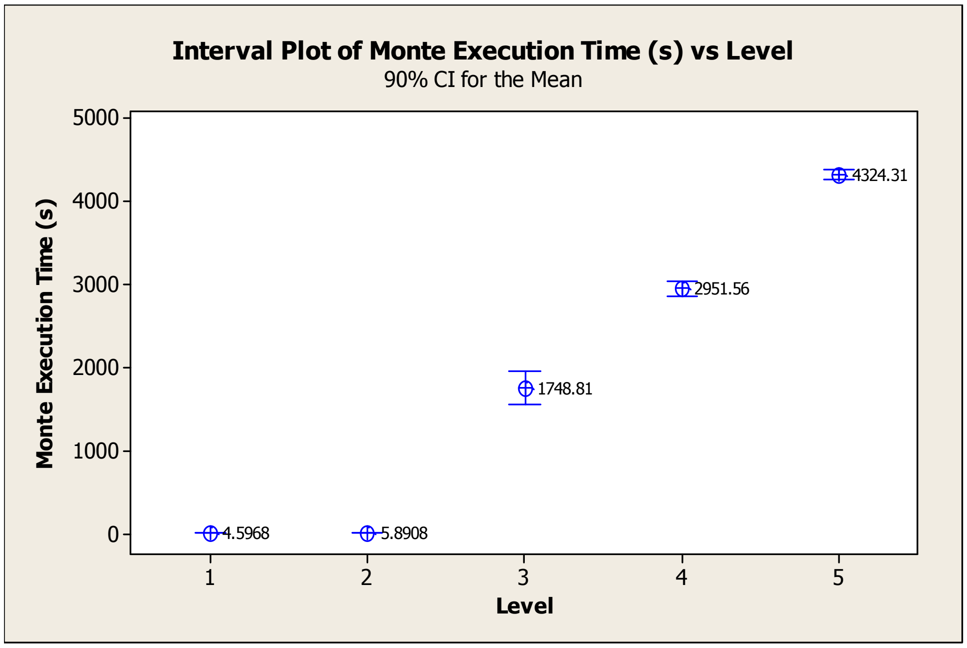


Figure A.12. Mean Interval Plot of Monte with OllyDbg and Optimization Off

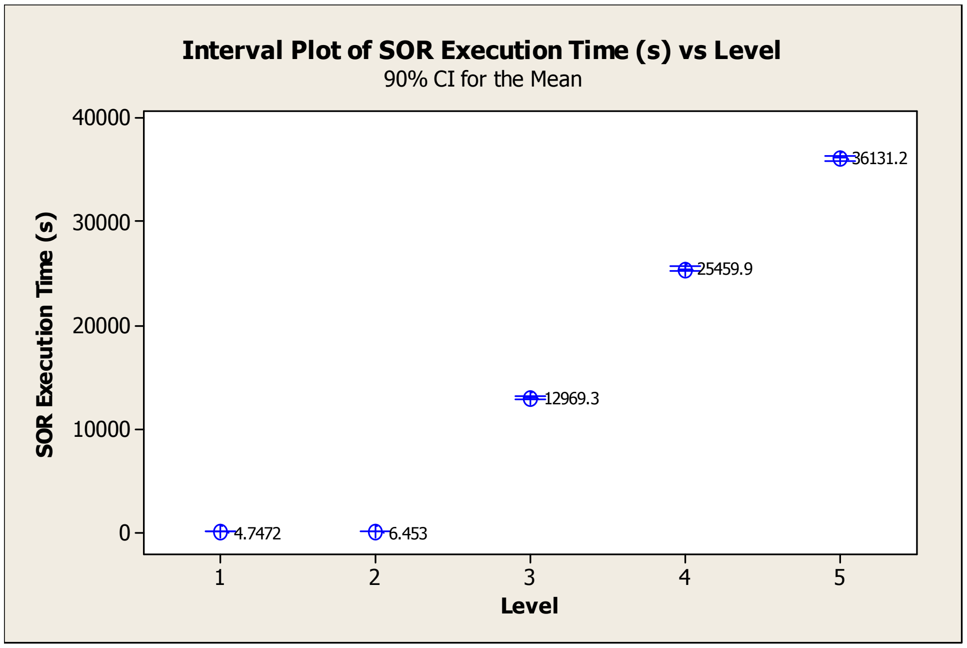


Figure A.13. Mean Interval Plot of SOR with OllyDbg and Optimization Off

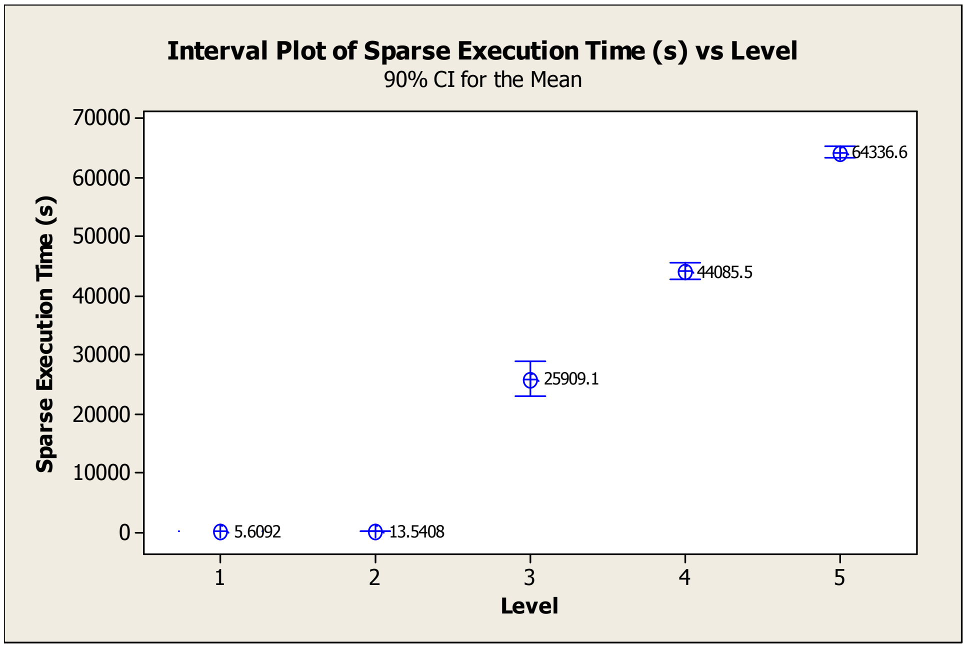


Figure A.14. Mean Interval Plot of Sparse with OllyDbg and Optimization Off

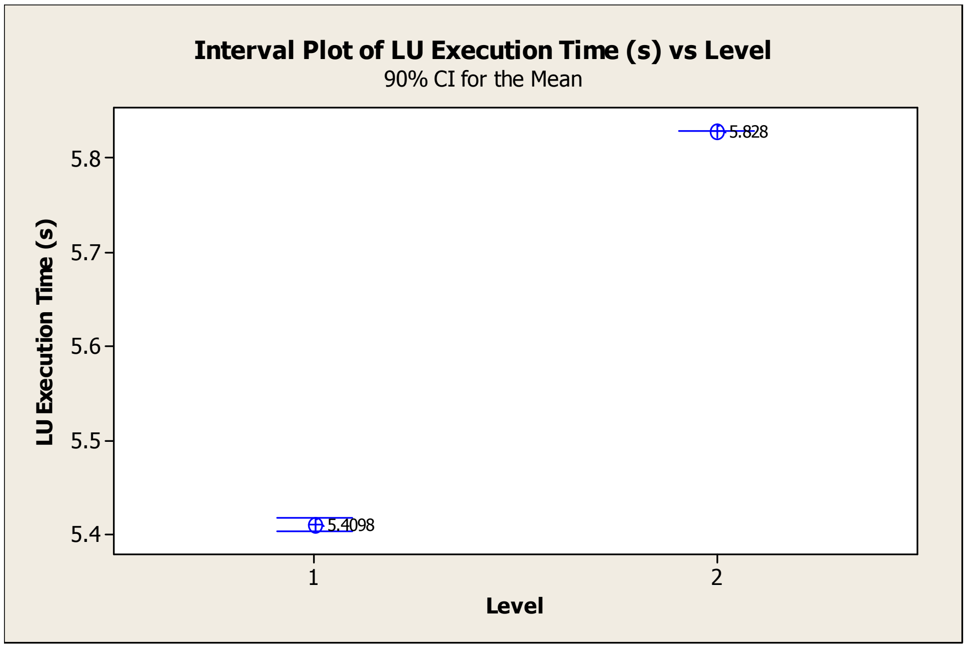


Figure A.15. Mean Interval Plot of LU with OllyDbg and Optimization Off Levels 1-2

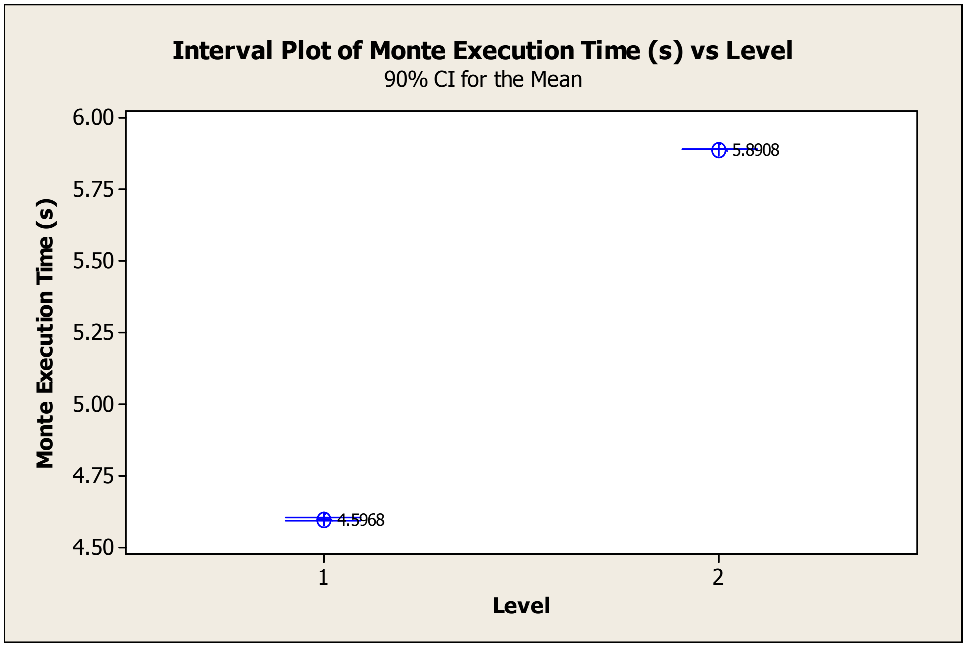


Figure A.16. Mean Interval Plot of Monte with OllyDbg and Optimization Off Levels 1-2

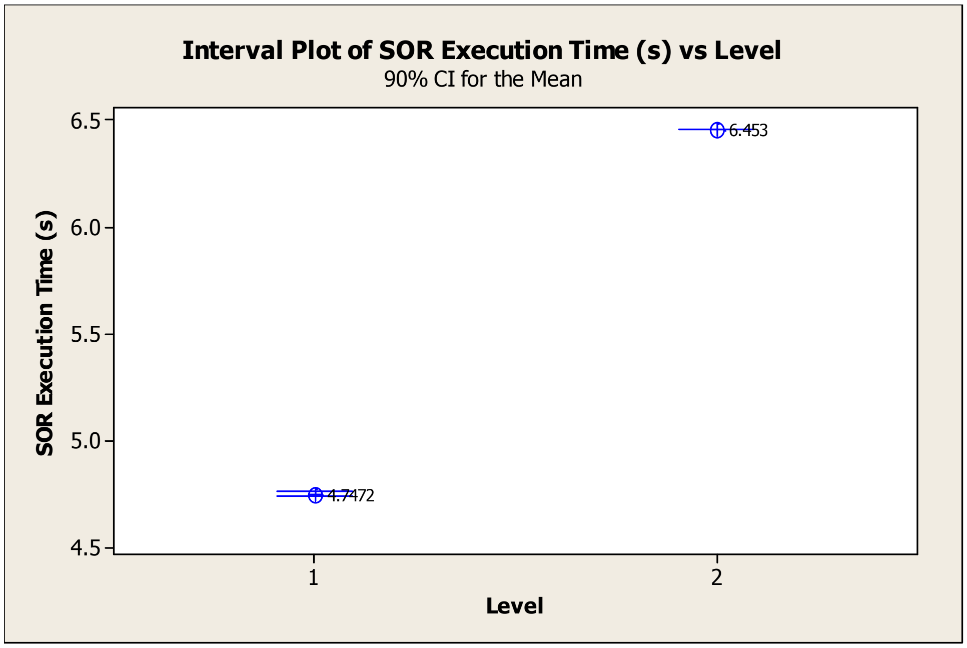


Figure A.17. Mean Interval Plot of SOR with OllyDbg and Optimization Off Levels 1-2

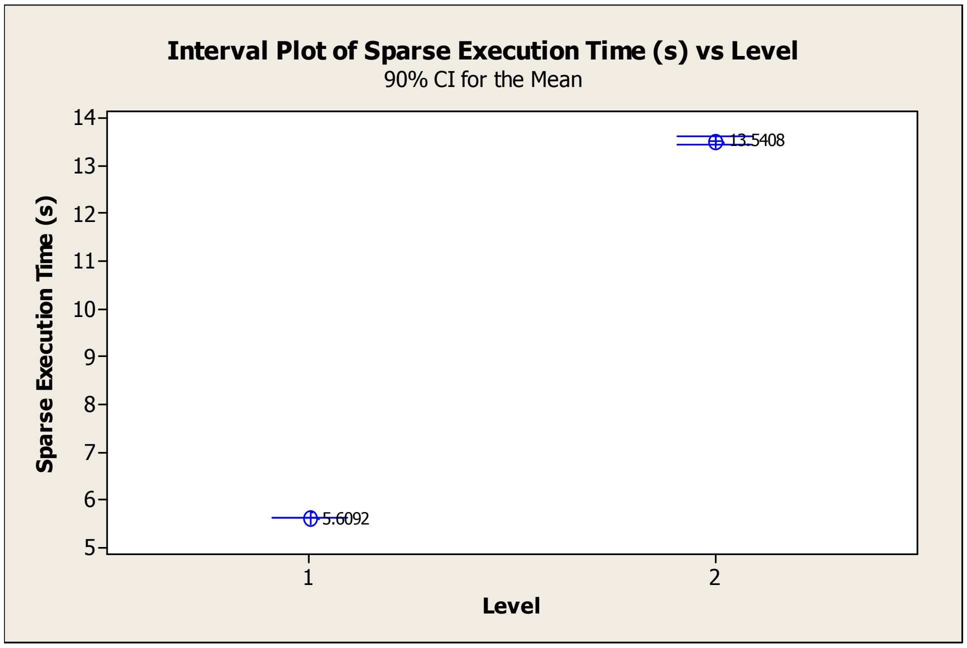


Figure A.18. Mean Interval Plot of Sparse with OllyDbg and Optimization Off Levels 1-2

```
The regression equation is
LU Execution Time (s) = - 348 + 245 Level

Predictor      Coef  SE Coef      T      P
Constant    -348.26    40.44  -8.61  0.000
Level        245.36    12.19  20.12  0.000

S = 86.2154   R-Sq = 94.6%   R-Sq(adj) = 94.4%

Analysis of Variance

Source          DF       SS       MS       F      P
Regression       1  3010161  3010161  404.97  0.000
Residual Error  23   170961     7433
Total           24  3181123
```

Figure A.19. Regression Model for LU with OllyDbg and Optimization Off, Levels 1-5

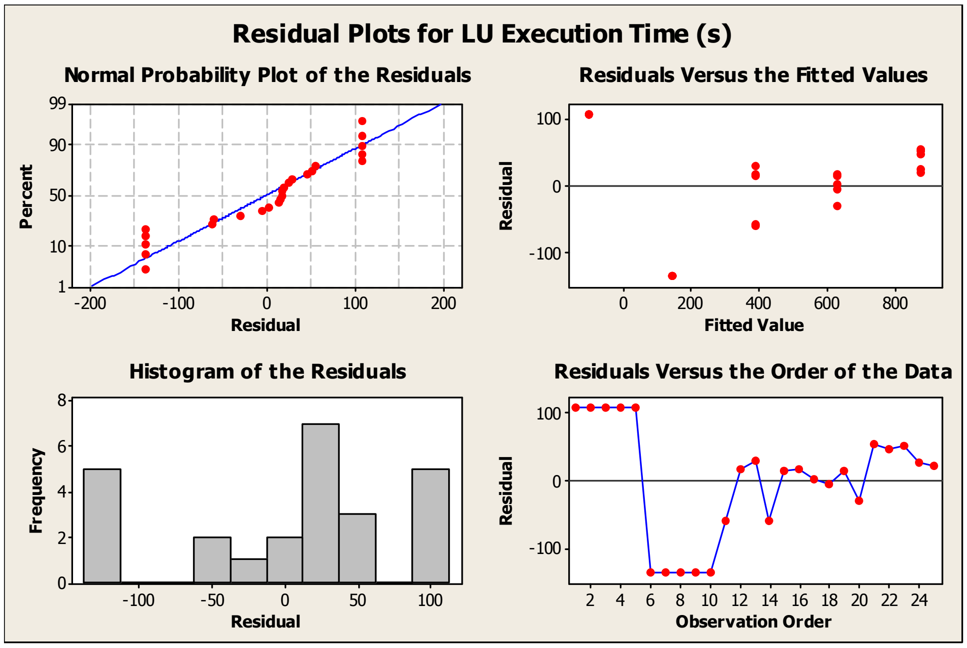


Figure A.20. 4-in-1 Plot for LU with OllyDbg and Optimization Off, Levels 1-5

```
The regression equation is
Monte Execution Time (s) = - 1668 + 1159 Level

Predictor      Coef  SE Coef      T      P
Constant    -1668.5    191.5  -8.71  0.000
Level       1158.51    57.75  20.06  0.000

S = 408.339   R-Sq = 94.6%   R-Sq(adj) = 94.4%

Analysis of Variance

Source          DF        SS        MS       F      P
Regression       1  67107139  67107139  402.46  0.000
Residual Error  23   3835036    166741
Total           24  70942175
```

Figure A.21. Regression Model for Monte w/ OllyDbg and Optimization Off, Levels 1-5

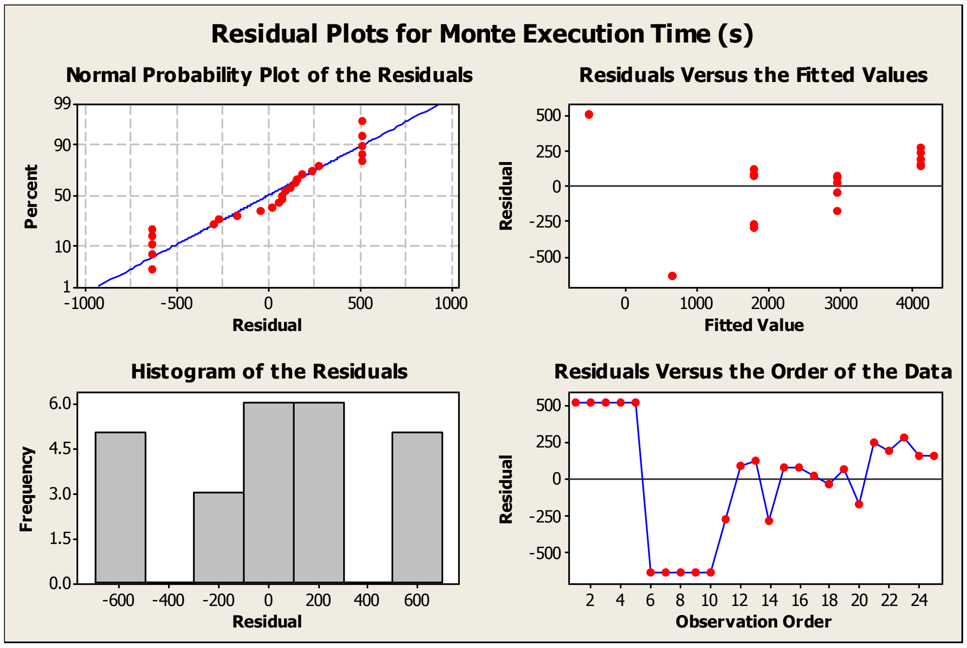


Figure A.22. 4-in-1 Plot for Monte with OllyDbg and Optimization Off, Levels 1-5

```
The regression equation is
SOR Opt-Off Execution Time (s) = - 14398 + 9771 Level

Predictor     Coef  SE Coef      T      P
Constant    -14398     1624  -8.86  0.000
Level       9770.6    489.7  19.95  0.000

S = 3462.89   R-Sq = 94.5%   R-Sq(adj) = 94.3%

Analysis of Variance

Source          DF          SS          MS       F      P
Regression       1  4773261136  4773261136  398.05  0.000
Residual Error  23   275806796    11991600
Total           24  5049067932
```

Figure A.23. Regression Model for SOR with OllyDbg and Optimization Off, Levels 1-5

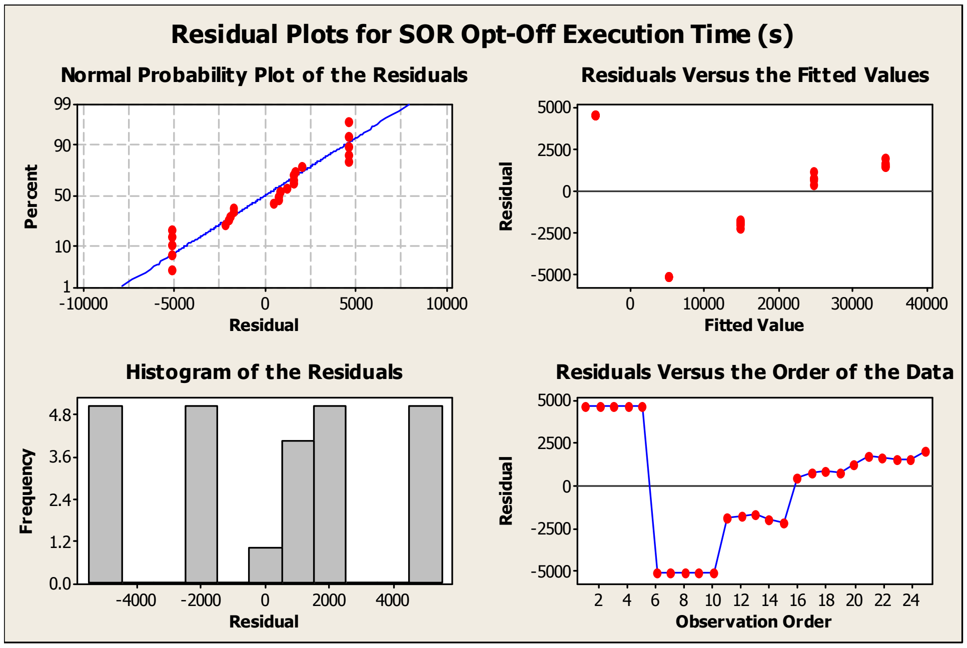


Figure A.24. 4-in-1 Plot for SOR with OllyDbg and Optimization Off, Levels 1-5

```
The regression equation is
Sparse Execution Time (s) = - 24950 + 17273 Level

Predictor      Coef  SE Coef      T      P
Constant     -24950     2854  -8.74  0.000
Level       17273.4    860.6  20.07  0.000

S = 6085.30   R-Sq = 94.6%   R-Sq(adj) = 94.4%

Analysis of Variance

Source          DF           SS           MS       F      P
Regression       1  14918497410  14918497410  402.87  0.000
Residual Error  23    851710254     37030881
Total           24  15770207664
```

Figure A.25. Regression Model for Sparse w/ OllyDbg and Optimization Off, Levels 1-5

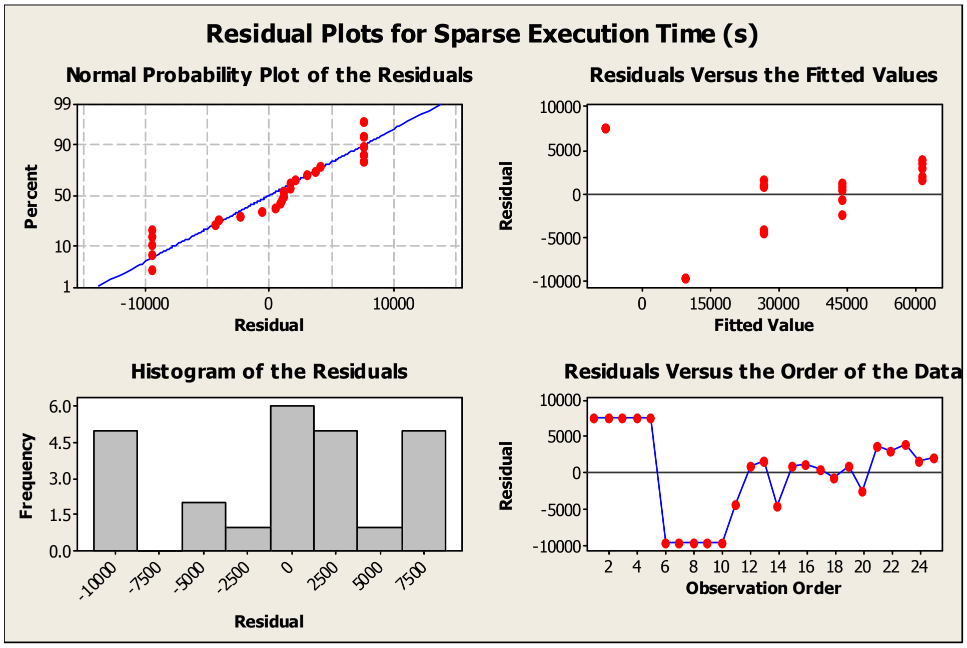


Figure A.26. 4-in-1 Plot for Sparse with OllyDbg and Optimization Off, Levels 1-5

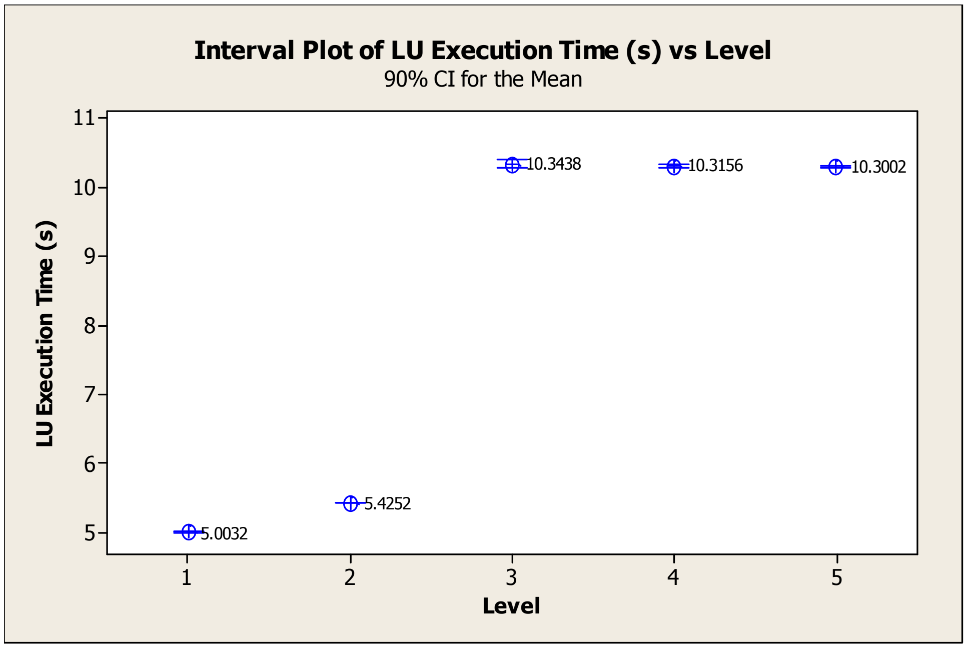


Figure A.27. Mean Interval Plot of LU with OllyDbg and Optimization On Levels 1-5

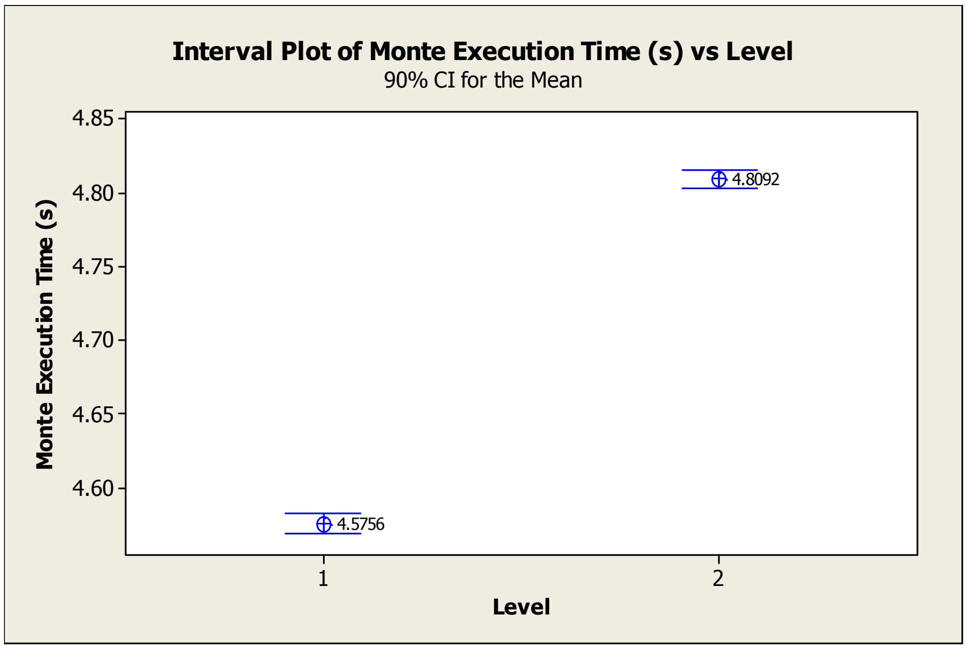


Figure A.28. Mean Interval Plot of Monte with OllyDbg and Optimization On Levels 1-2

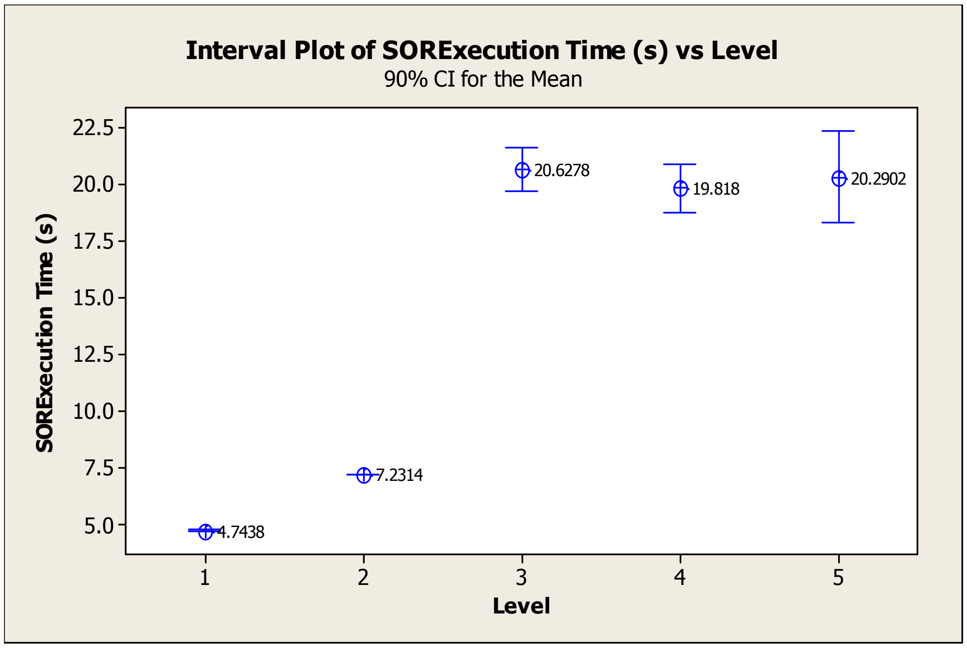


Figure A.29. Mean Interval Plot of SOR with OllyDbg and Optimization On Levels 1-5

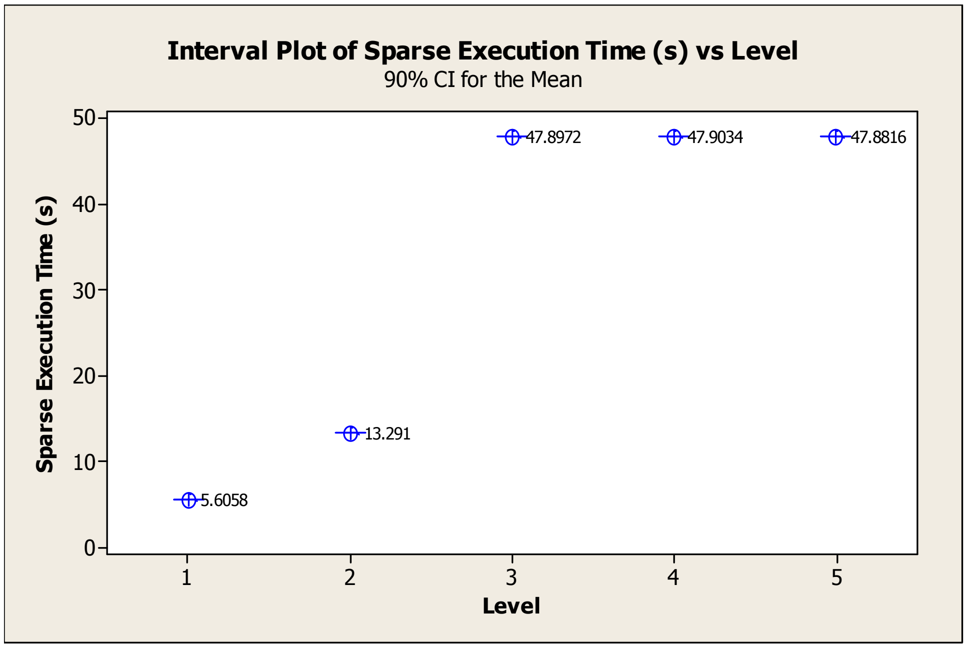


Figure A.30. Mean Interval Plot of Sparse with OllyDbg and Optimization On Levels 1-5

```
The regression equation is
LU (Execution Time) = 3.63 + 1.55 Level

Predictor    Coef   SE Coef     T      P
Constant   3.6323    0.5948  6.11  0.000
Level      1.5484    0.1793  8.63  0.000

S = 1.26819   R-Sq = 76.4%   R-Sq(adj) = 75.4%

Analysis of Variance

Source          DF      SS      MS      F      P
Regression       1  119.88  119.88  74.54  0.000
Residual Error  23   36.99    1.61
Total           24  156.87
```

Figure A.31. Regression Model for LU w/ OllyDbg and Optimization On, Levels 1-5

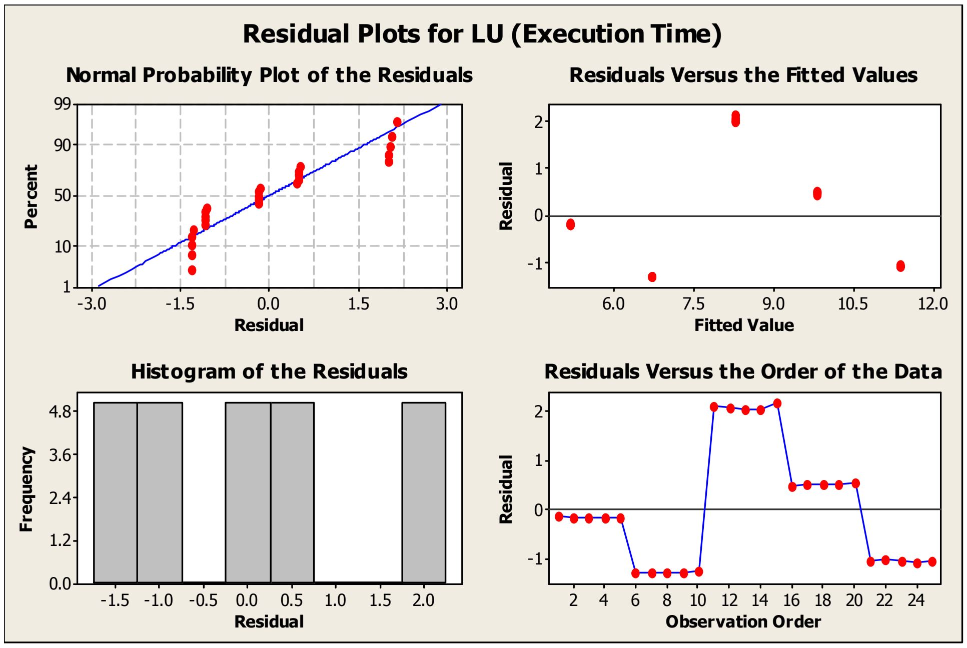


Figure A.32. 4-in-1 Plot for LU with OllyDbg and Optimization On, Levels 1-5

```
The regression equation is
Monte (Execution Time) = 4.34 + 0.234 Level

Predictor       Coef    SE Coef        T      P
Constant     4.34200    0.00695   625.09  0.000
Level       0.233600   0.004393    53.17  0.000

S = 0.00694622   R-Sq = 99.7%   R-Sq(adj) = 99.7%

Analysis of Variance

Source          DF        SS        MS        F      P
Regression       1   0.13642   0.13642  2827.41  0.000
Residual Error   8   0.00039   0.00005
Total            9   0.13681

Unusual Observations

                 Monte
              (Execution
Obs  Level       Time)      Fit    SE Fit  Residual  St Resid
  3   1.00     4.56300  4.57560  0.00311  -0.01260    -2.03R

R denotes an observation with a large standardized residual.
```

Figure A.33. Regression Model for Monte w/ OllyDbg and Optimization On, Levels 1-5

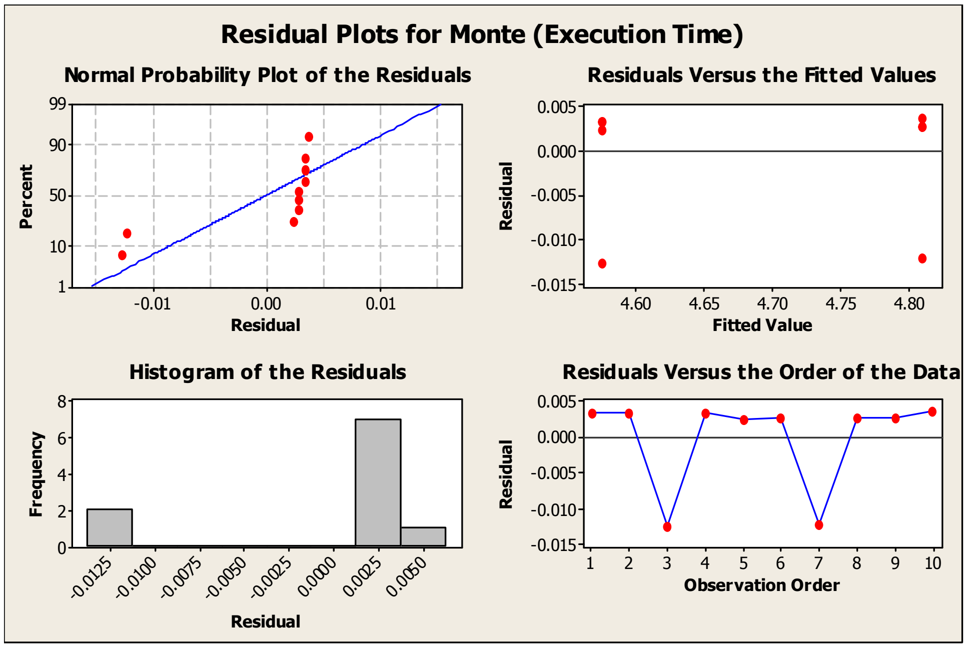


Figure A.34. 4-in-1 Plot for Monte with OllyDbg and Optimization On, Levels 1-5

```
The regression equation is
SOR (Execution Time) = 1.44 + 4.37 Level

Predictor     Coef   SE Coef     T      P
Constant     1.438     1.720  0.84  0.412
Level       4.3679    0.5187  8.42  0.000

S = 3.66802   R-Sq = 75.5%   R-Sq(adj) = 74.4%

Analysis of Variance

Source          DF       SS      MS      F      P
Regression       1   953.94  953.94  70.90  0.000
Residual Error  23   309.45   13.45
Total           24  1263.39

Unusual Observations

                   SOR
             (Execution
Obs  Level        Time)     Fit  SE Fit  Residual  St Resid
 13   3.00       22.050  14.542   0.734     7.508     2.09R

R denotes an observation with a large standardized residual.
```

Figure A.35. Regression Model for SOR w/ OllyDbg and Optimization On, Levels 1-5

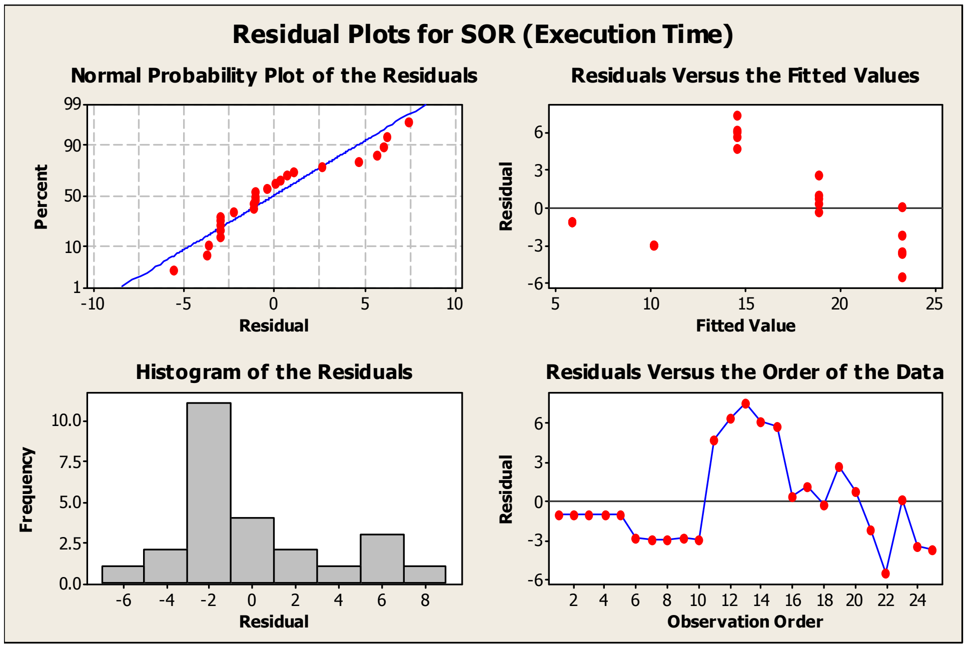


Figure A.36. 4-in-1 Plot for SOR with OllyDbg and Optimization On, Levels 1-5

```
The regression equation is
Sparse (Execution Time) = - 3.23 + 11.9 Level

Predictor     Coef  SE Coef      T      P
Constant    -3.233    4.281  -0.76  0.458
Level       11.916    1.291   9.23  0.000

S = 9.12728   R-Sq = 78.7%   R-Sq(adj) = 77.8%

Analysis of Variance

Source          DF      SS      MS      F      P
Regression       1  7100.0  7100.0  85.23  0.000
Residual Error  23  1916.1    83.3
Total           24  9016.1
```

Figure A.37. Regression Model for Sparse w/ OllyDbg and Optimization On, Levels 1-5

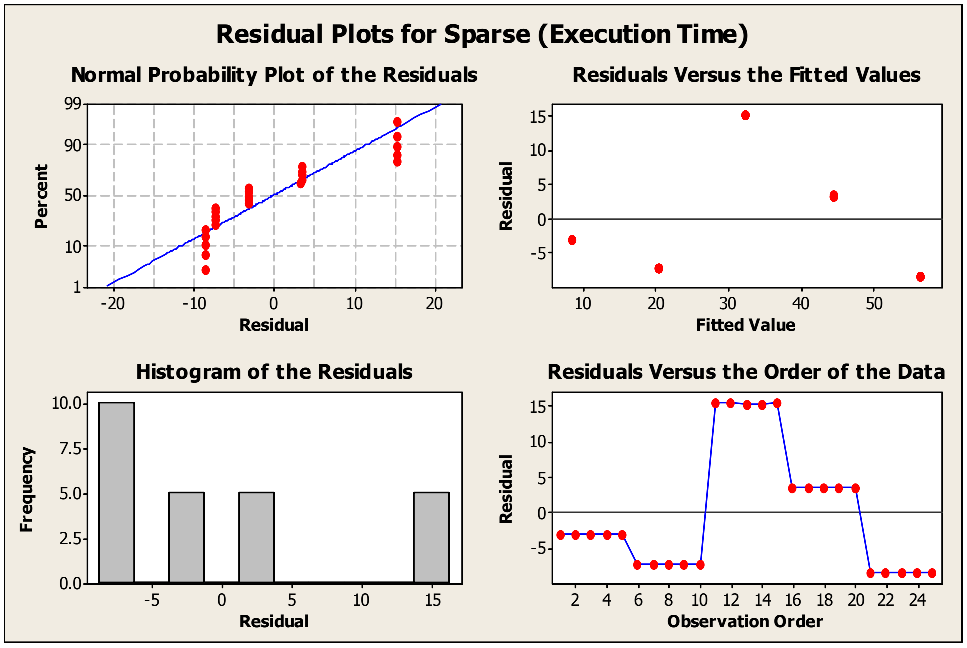


Figure A.38. 4-in-1 Plot for Sparse with OllyDbg and Optimization On, Levels 1-5

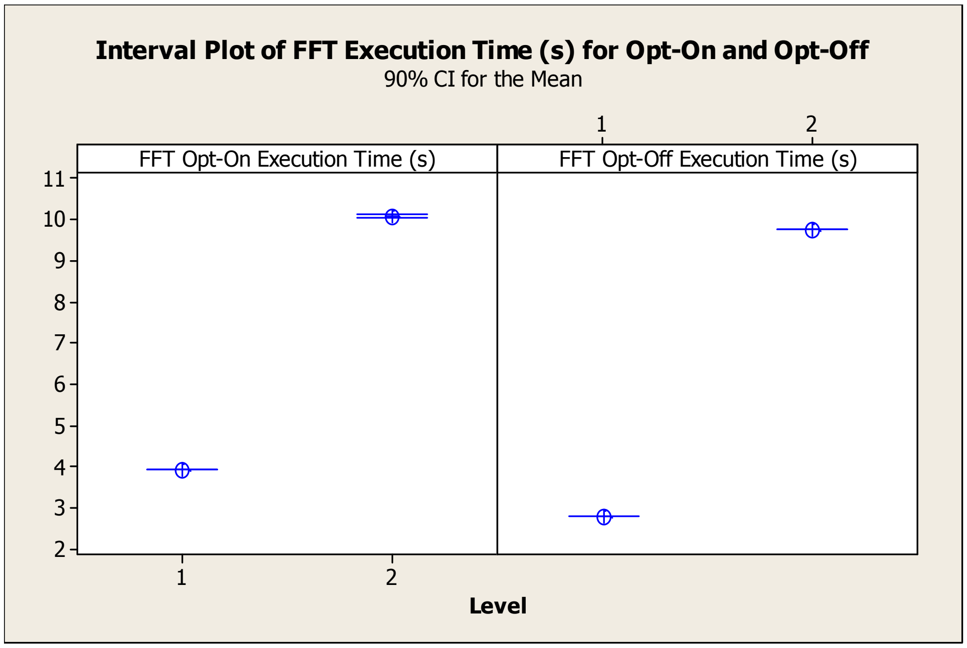


Figure A.39. Mean Interval Plot of FFT with Optimization On and Off with OllyDbg, Levels 1-2

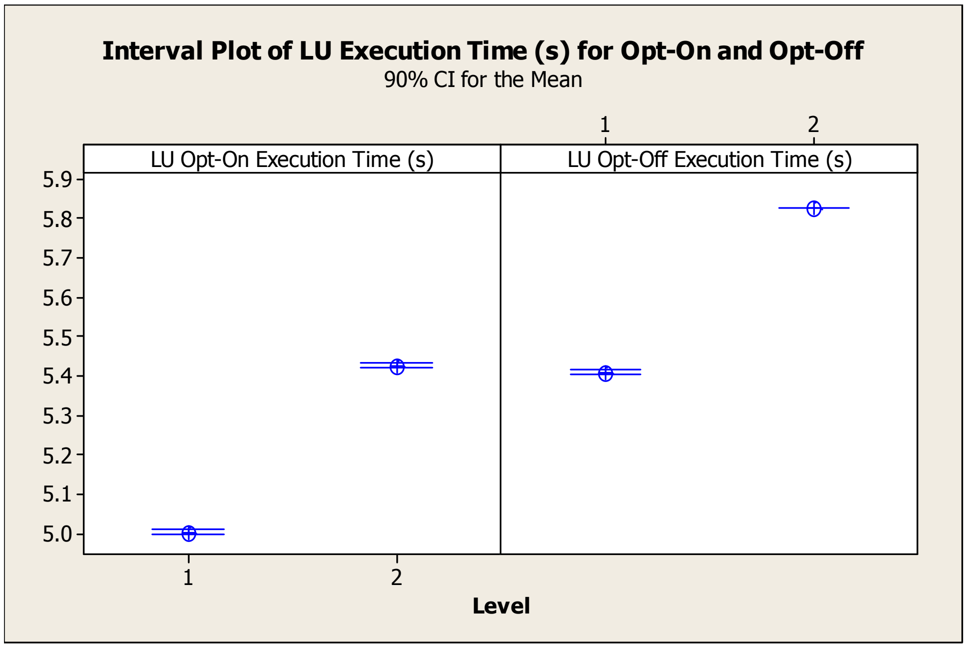


Figure A.40. Mean Interval Plot of LU with Optimization On and Off with OllyDbg, Levels 1-2

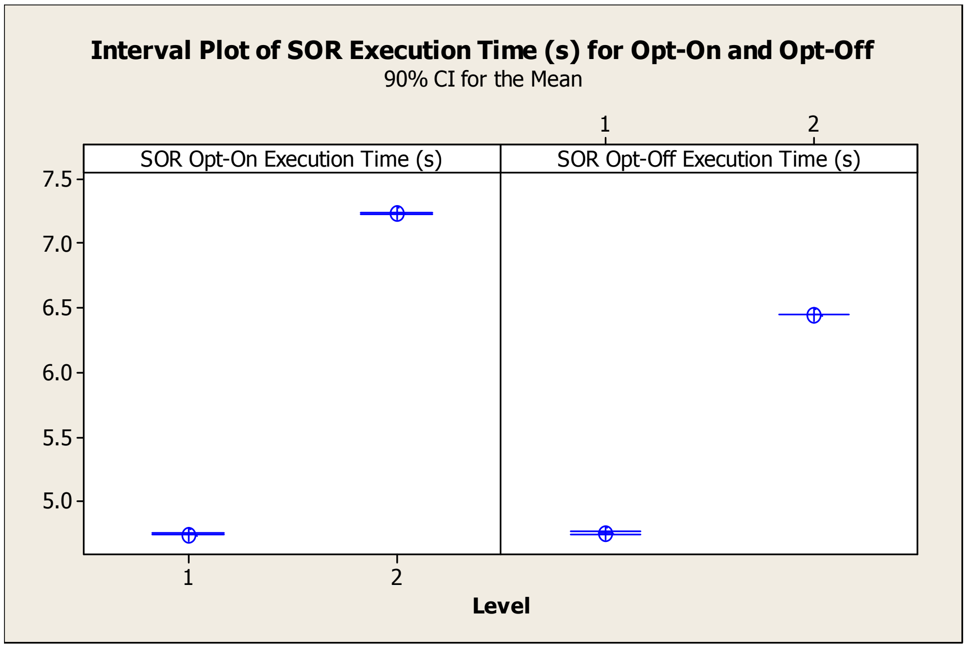


Figure A.41. Mean Interval Plot of SOR with Optimization On and Off with OllyDbg, Levels 1-2

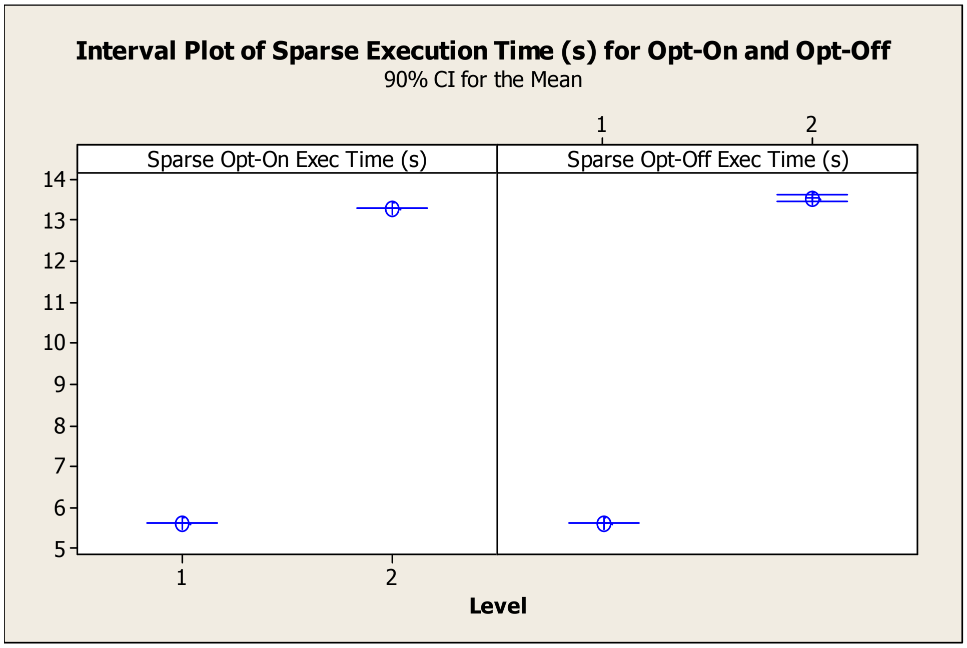


Figure A.42. Mean Interval Plot of Sparse with Optimization On and Off with OllyDbg, Levels 1-2

Table A.1. Ggrep with Optimization Off and IDAPro, Levels 1-5

Level	Test Expression	Mean	St-Dev	90% Confidence Interval
1 (baseline)	Expression 1	18.394	0.008	[18.386, 18.402]
2 (hidden function in use)	Expression 1	18.462	0.049	[18.416, 18.509]
3 (hidden function w/ 4 threads)	Expression 1	18.36	0.131	[18.235, 18.485]
4 (hidden function w/ 8 threads)	Expression 1	18.394	0.087	[18.311, 18.477]
5 (hidden function w/ 12 threads)	Expression 1	18.378	0.086	[18.295, 18.46]
1 (baseline)	Expression 2	5.3904	0.0005	[5.3899, 5.3909]
2 (hidden function in use)	Expression 2	5.3874	0.0069	[5.3808, 5.3940]
3 (hidden function w/ 4 threads)	Expression 2	5.3686	0.0088	[5.3602, 5.3770]
4 (hidden function w/ 8 threads)	Expression 2	5.3752	0.0153	[5.3607, 5.3897]
5 (hidden function w/ 12 threads)	Expression 2	5.369	0.0082	[5.3612, 5.3768]
1 (baseline)	Expression 3	219.1	0.12	[218.98, 219.21]
2 (hidden function in use)	Expression 3	218.98	1.43	[217.62, 220.35]
3 (hidden function w/ 4 threads)	Expression 3	217.22	0.09	[217.13, 217.30]
4 (hidden function w/ 8 threads)	Expression 3	217.27	0.08	[217.20, 217.34]
5 (hidden function w/ 12 threads)	Expression 3	217.38	0.39	[217.01, 217.76]

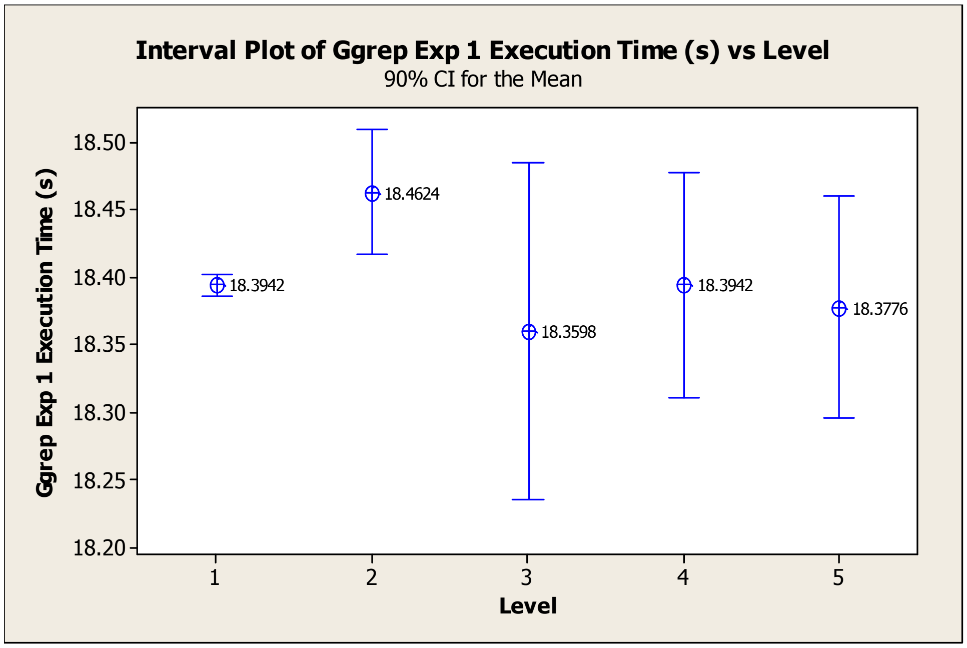


Figure A.43. Mean Interval Plot of Ggrep Expression 1 w/ IDAPro and Optimization Off

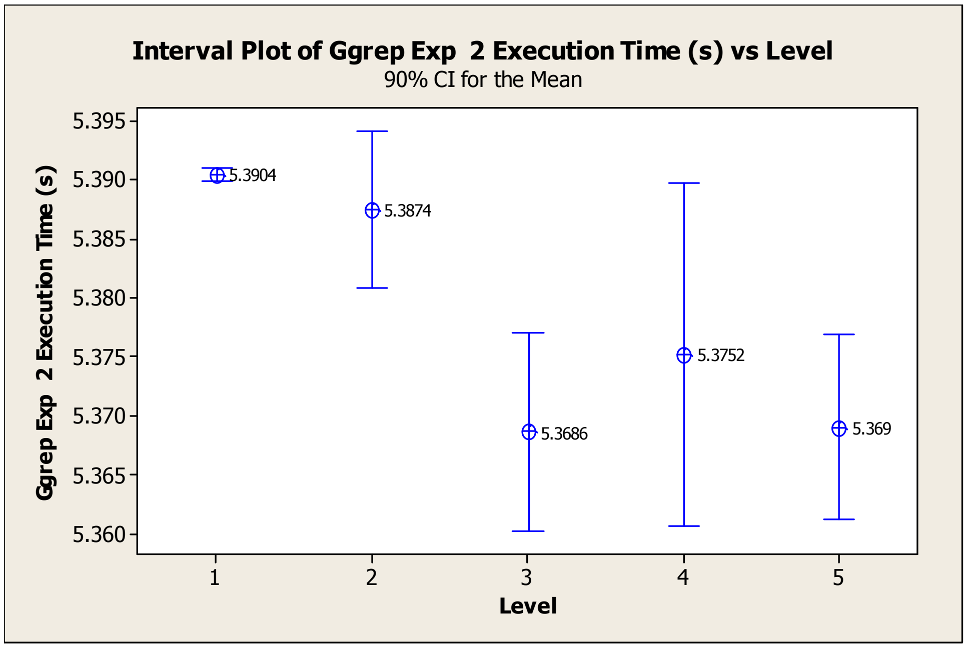


Figure A.44. Mean Interval Plot of Ggrep Expression 2 w/ IDAPro and Optimization Off

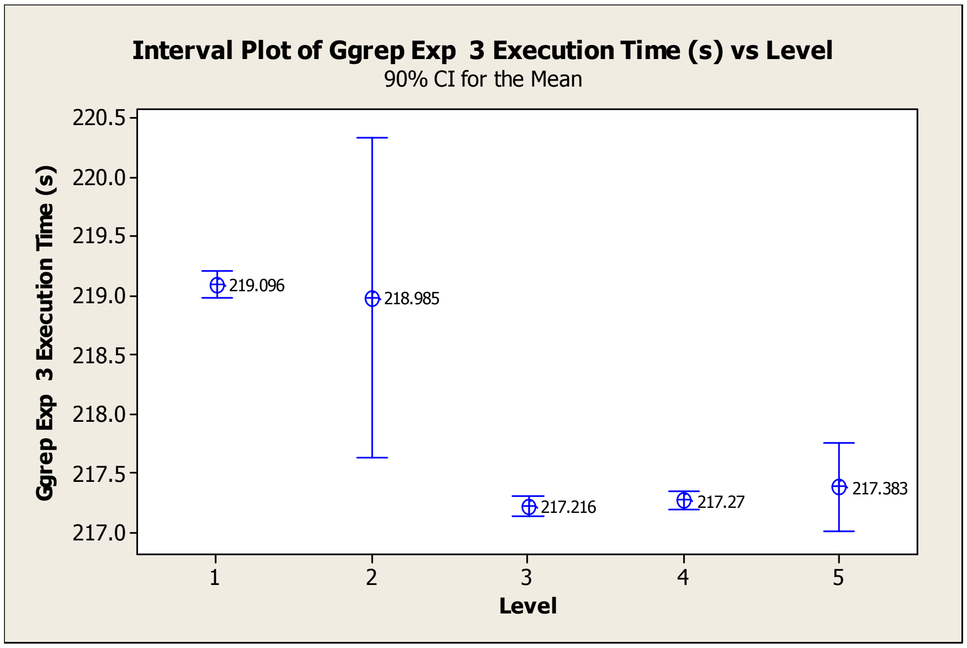


Figure A.45. Mean Interval Plot of Ggrep Expression 3 w/ IDAPro and Optimization Off

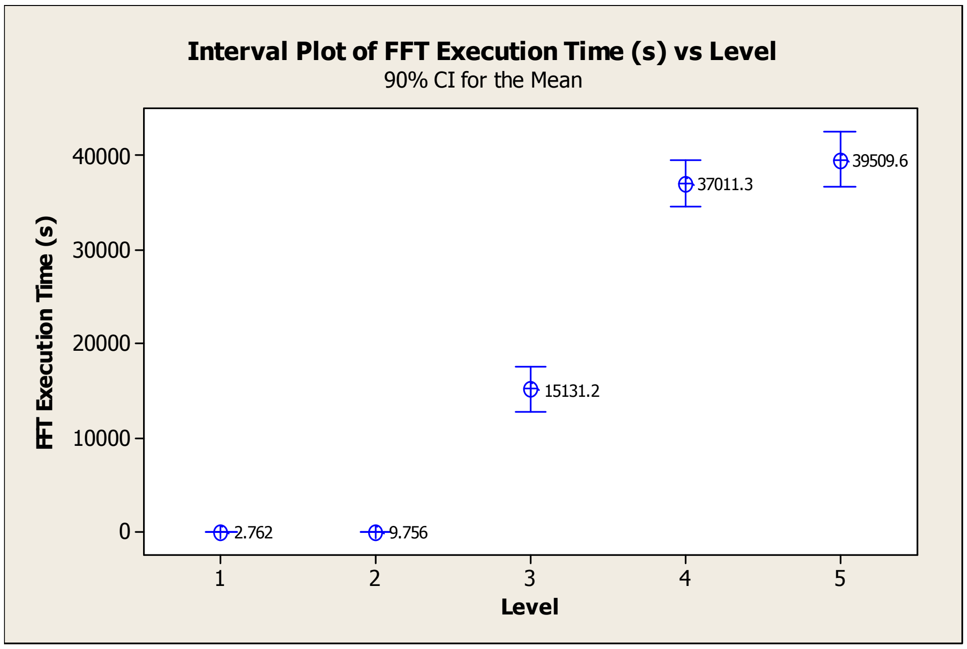


Figure A.46. Mean Interval Plot of FFT with IDAPro and Optimization Off

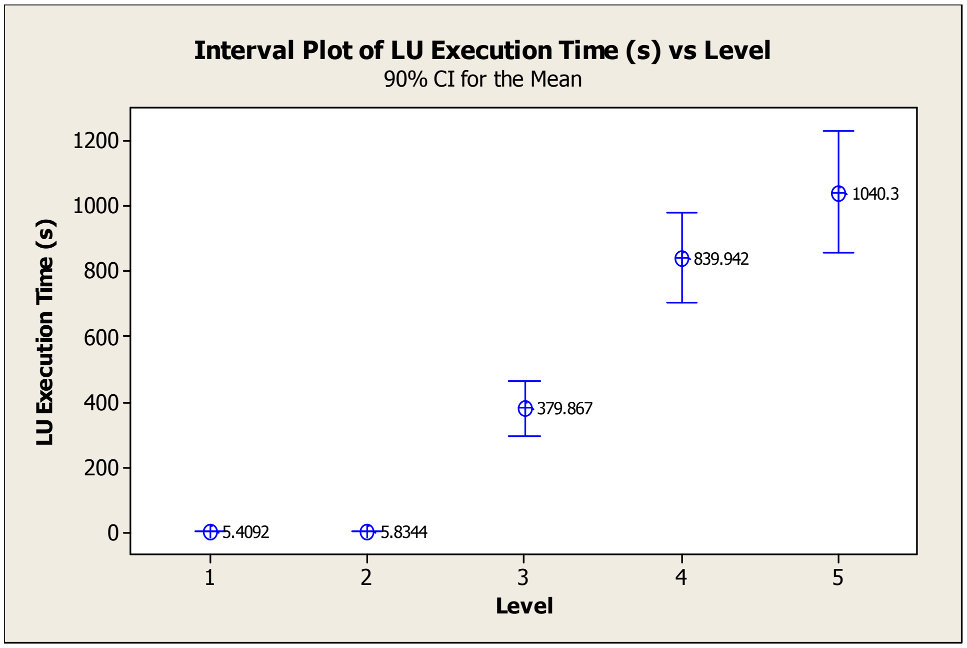


Figure A.47. Mean Interval Plot of LU with IDAPro and Optimization Off

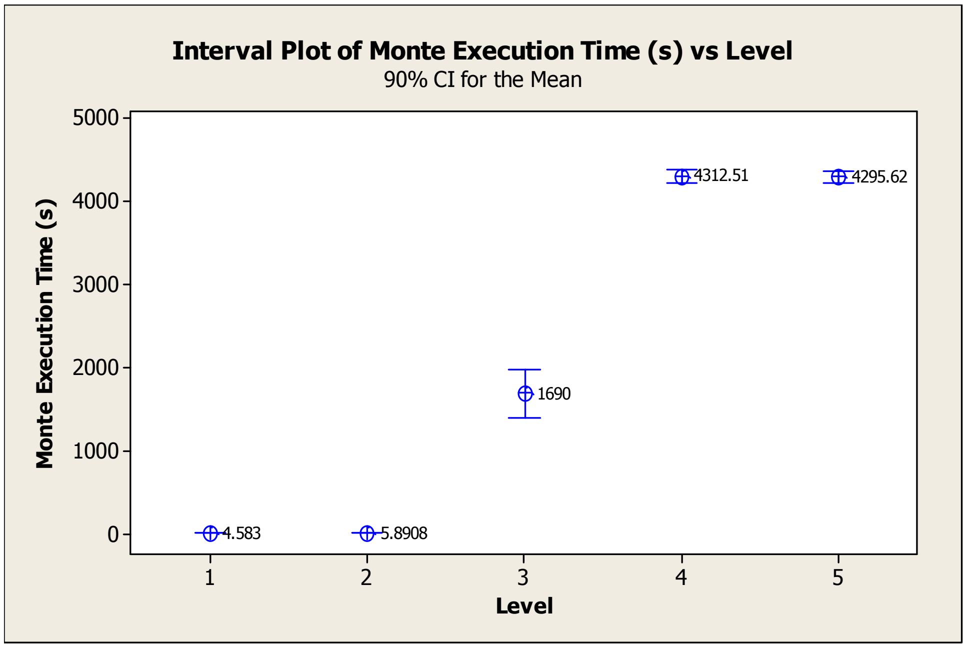


Figure A.48. Mean Interval Plot of Monte with IDAPro and Optimization Off

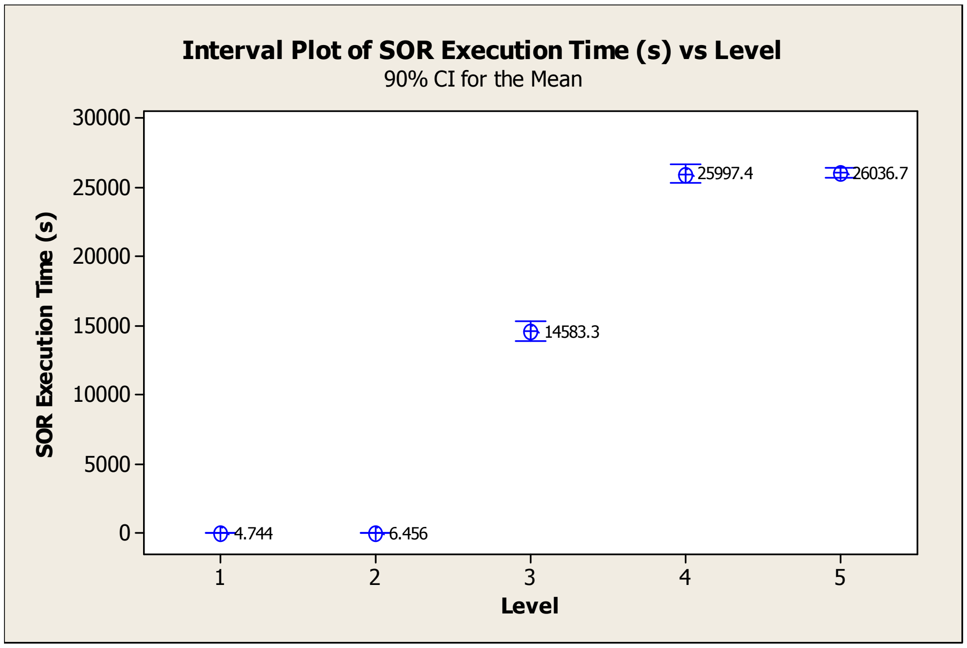


Figure A.49. Mean Interval Plot of SOR with IDAPro and Optimization Off

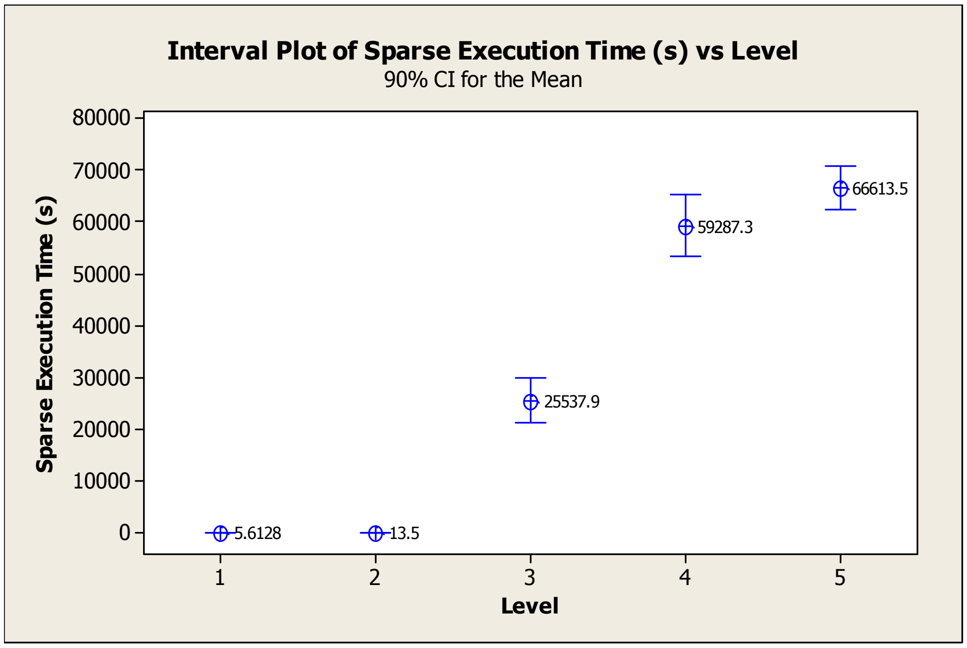


Figure A.50. Mean Interval Plot of Sparse with IDAPro and Optimization Off

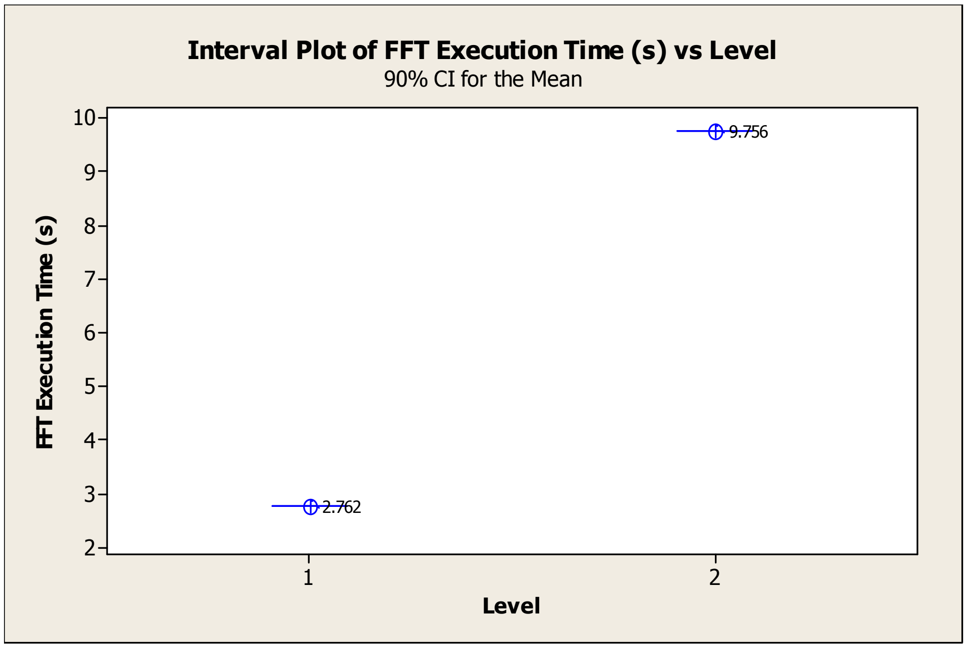


Figure A.51. Mean Interval Plot of FFT with IDAPro and Optimization Off, Levels 1 and 2

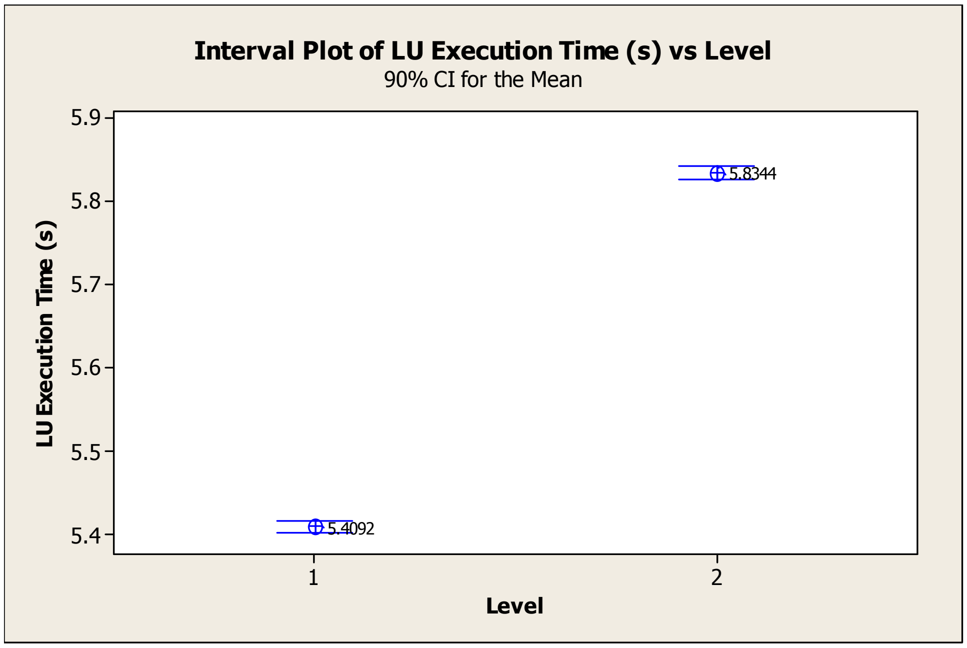


Figure A.52. Mean Interval Plot of LU with IDAPro and Optimization Off, Levels 1 and 2

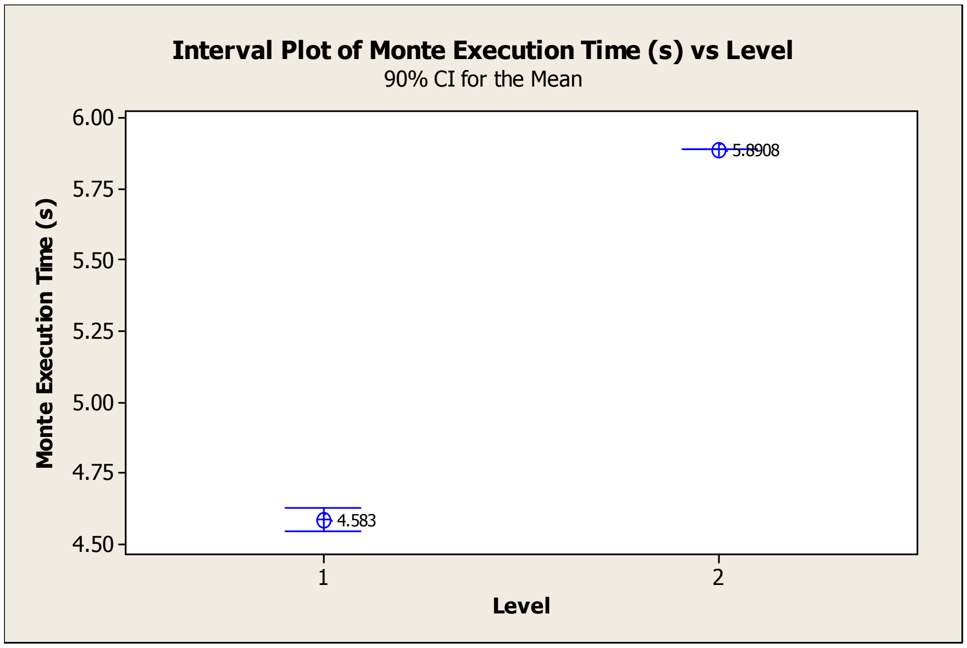


Figure A.53. Mean Interval Plot of Monte with IDAPro and Optimization Off, Levels 1 and 2

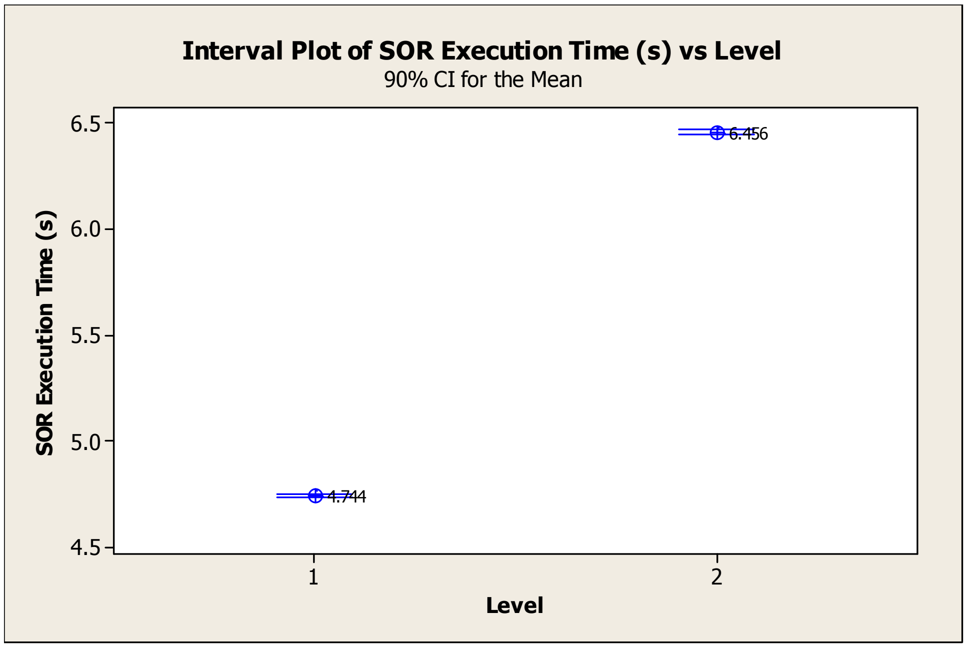


Figure A.54. Mean Interval Plot of SOR with IDAPro and Optimization Off, Levels 1 and 2

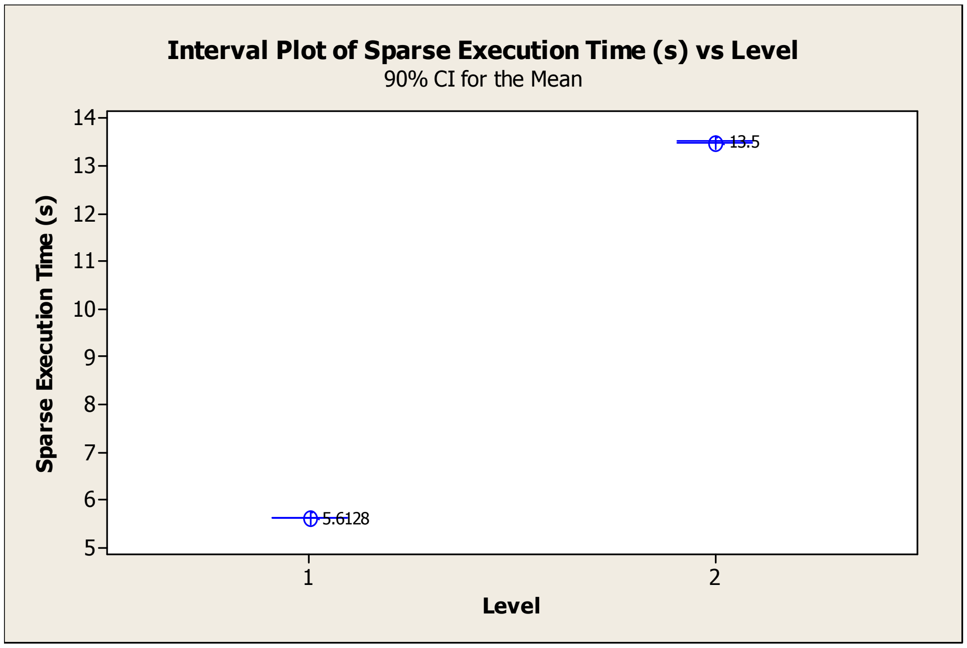


Figure A.55. Mean Interval Plot of Sparse with IDAPro and Optimization Off, Levels 1 and 2

```
The regression equation is
Ggrep Exp 1 (Execution Time) = 18.4 - 0.0101 Level

Predictor       Coef   SE Coef       T      P
Constant     18.4281    0.0396  465.42  0.000
Level       -0.01014   0.01194   -0.85  0.404

S = 0.0844158   R-Sq = 3.0%   R-Sq(adj) = 0.0%

Analysis of Variance

Source          DF        SS        MS     F      P
Regression       1  0.005141  0.005141  0.72  0.404
Residual Error  23  0.163899  0.007126
Total           24  0.169040

Unusual Observations

             Ggrep Exp
                     1
            (Execution
Obs  Level       Time)      Fit  SE Fit  Residual  St Resid
 12   3.00     18.5880  18.3976  0.0169    0.1904     2.30R

R denotes an observation with a large standardized residual.
```

Figure A.56. Regression Model for Ggrep Expression 1 w/ IDAPro and Optimization Off, Levels 1-5

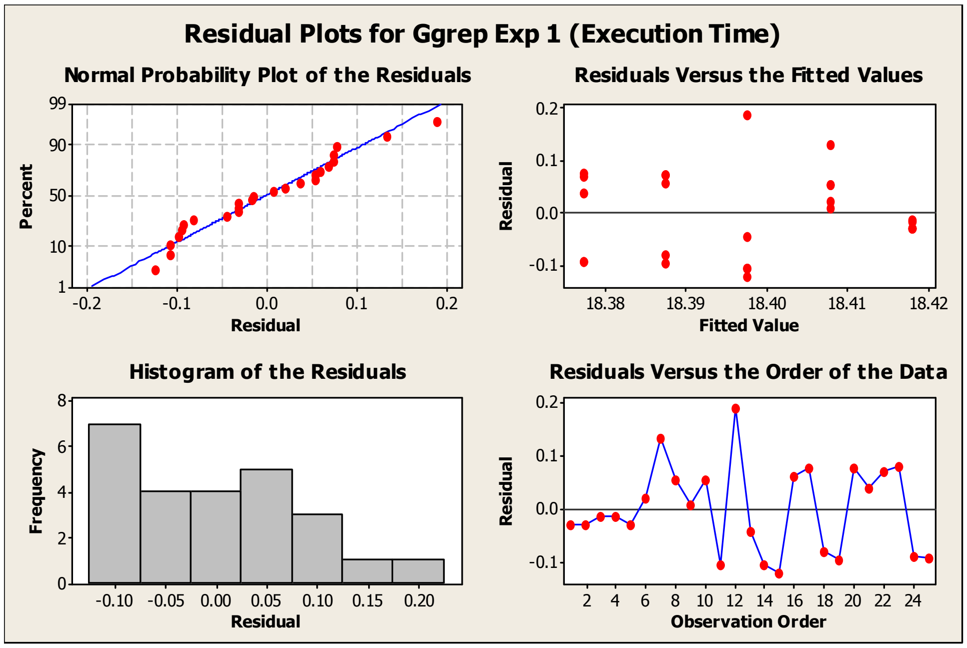


Figure A.57. 4-in-1 Plot for Ggrep Expression 1 with IDAPro and Optimization Off, Levels 1-5

```
The regression equation is
Ggrep Exp  2 (Execution Time) = 5.39 - 0.00550 Level

Predictor       Coef   SE Coef        T      P
Constant     5.39462   0.00468  1153.62  0.000
Level      -0.005500  0.001410    -3.90  0.001

S = 0.00996982   R-Sq = 39.8%   R-Sq(adj) = 37.2%

Analysis of Variance

Source          DF         SS         MS      F      P
Regression       1  0.0015125  0.0015125  15.22  0.001
Residual Error  23  0.0022861  0.0000994
Total           24  0.0037986
```

Figure A.58. Regression Model for Ggrep Expression 2 w/ IDAPro and Optimization Off, Levels 1-5

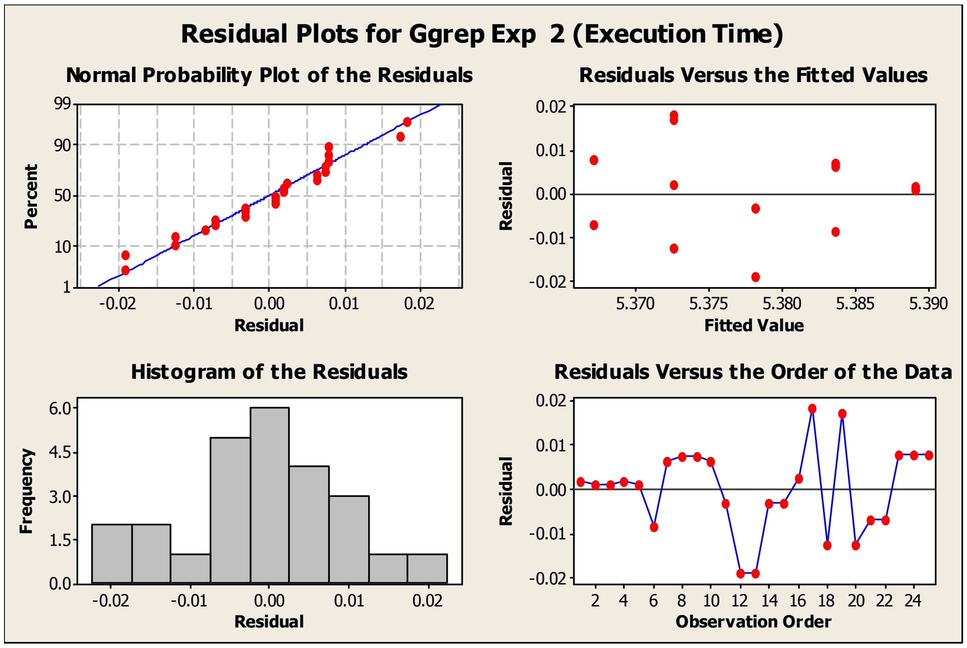


Figure A.59. 4-in-1 Plot for Ggrep Expression 2 with IDAPro and Optimization Off, Levels 1-5

```
The regression equation is
Ggrep Exp  3 (Execution Time) = 220 - 0.514 Level

Predictor      Coef  SE Coef       T      P
Constant    219.532    0.368  596.60  0.000
Level       -0.5141   0.1109   -4.63  0.000

S = 0.784526   R-Sq = 48.3%   R-Sq(adj) = 46.0%

Analysis of Variance

Source          DF      SS      MS      F      P
Regression       1  13.215  13.215  21.47  0.000
Residual Error  23  14.156   0.615
Total           24  27.371

Unusual Observations

             Ggrep Exp
                     3
            (Execution
Obs  Level       Time)     Fit  SE Fit  Residual  St Resid
  8   2.00     221.531 218.504   0.192     3.027     3.98R

R denotes an observation with a large standardized residual.
```

Figure A.60. Regression Model for Ggrep Expression 3 w/ IDAPro and Optimization Off, Levels 1-5

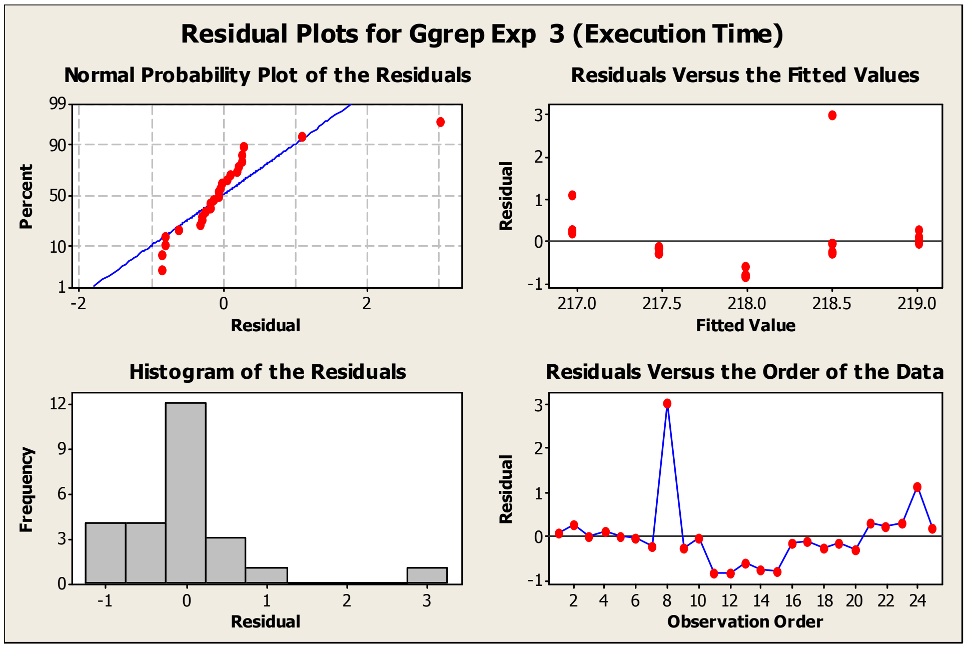


Figure A.61. 4-in-1 Plot for Ggrep Expression 3 with IDAPro and Optimization Off, Levels 1-5

Table A.2. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 1 with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	NO	NO	NO
2 (hidden function in use)	X	X	YES	NO	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.3. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 2 with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	YES	YES	YES
2 (hidden function in use)	X	X	YES	NO	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.4. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 3 with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.5. SciMark2 with Optimization Off and IDAPro, Levels 1-5

Level	Function	Mean	St-Dev	90% Confidence Interval
1 (baseline)	FFT	2.762	0.0067	[2.7556, 2.7684]
2 (hidden function in use)	FFT	9.756	0.0082	[9.7482, 9.7638]
3 (hidden function w/ 4 threads)	FFT	15131	2568	[12683, 17580]
4 (hidden function w/ 8 threads)	FFT	37011	2564	[34567, 39456]
5 (hidden function w/ 12 threads)	FFT	39510	3081	[36572, 42447]
1 (baseline)	LU	5.4092	0.0072	[5.4024, 5.410]
2 (hidden function in use)	LU	5.8344	0.0088	[5.8260, 5.8428]
3 (hidden function w/ 4 threads)	LU	379.87	87.71	[296.24, 463.49]
4 (hidden function w/ 8 threads)	LU	839.94	142.5	[704.08, 975.80]
5 (hidden function w/ 12 threads)	LU	1040.3	195.2	[854.2, 1226.4]
1 (baseline)	Monte	4.583	0.0435	[4.515, 5.6245]
2 (hidden function in use)	Monte	5.8908	0.0004	[5.8904, 5.8910]
3 (hidden function w/ 4 threads)	Monte	1690	305.8	[1398.4, 1981.6]
4 (hidden function w/ 8 threads)	Monte	4312.5	85.6	[4230.9, 4394.1]
5 (hidden function w/ 12 threads)	Monte	4295.6	67.1	[4231.6, 4359.6]
1 (baseline)	SOR	4.744	0.0082	[4.7362, 4.7518]
2 (hidden function in use)	SOR	6.456	0.0125	[6.4440, 6.4680]
3 (hidden function w/ 4 threads)	SOR	14583	737	[13881, 15286]
4 (hidden function w/ 8 threads)	SOR	25997	706	[25324, 26671]
5 (hidden function w/ 12 threads)	SOR	26037	400	[25656, 26418]
1 (baseline)	Sparse	5.6128	0.0129	[5.6005, 5.6251]
2 (hidden function in use)	Sparse	13.5	0.011	[13.489, 13.511]
3 (hidden function w/ 4 threads)	Sparse	25538	4428	[21317, 29759]
4 (hidden function w/ 8 threads)	Sparse	59287	6330	[53252, 65323]
5 (hidden function w/ 12 threads)	Sparse	66613	4345	[62471, 70756]

```
The regression equation is
FFT (Execution Time) = - 16472 + 11602 Level

Predictor      Coef   SE Coef      T      P
Constant     -16472      2692  -6.12  0.000
Level       11601.5     811.6  14.29  0.000

S = 5738.77   R-Sq = 89.9%   R-Sq(adj) = 89.4%

Analysis of Variance

Source          DF          SS          MS       F      P
Regression       1  6729778258  6729778258  204.34  0.000
Residual Error  23   757470477    32933499
Total           24  7487248735
```

Figure A.62. Regression Model for FFT w/ IDAPro and Optimization Off, Levels 1-5

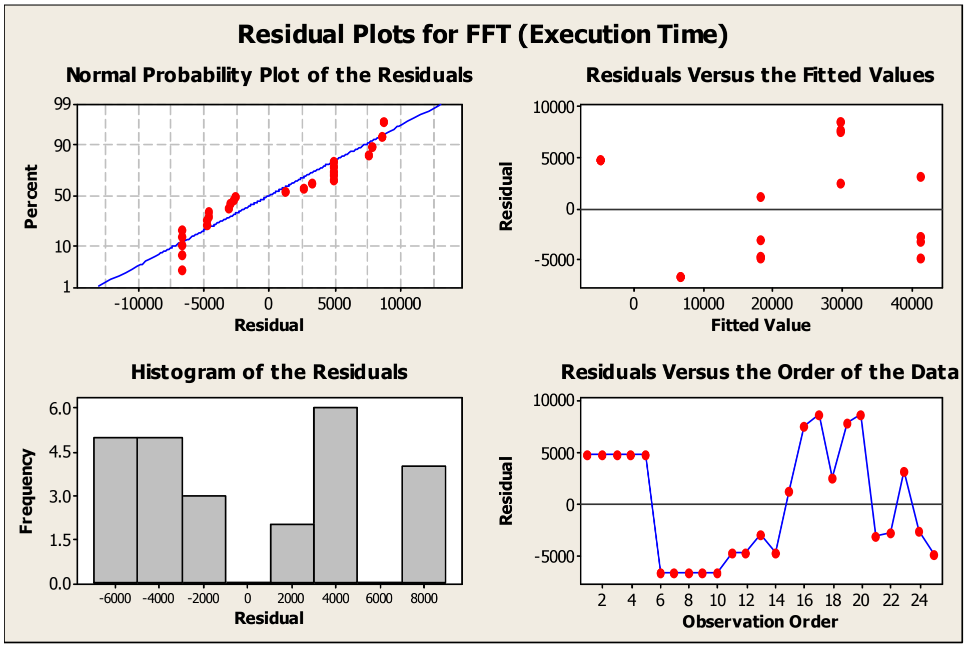


Figure A.63. 4-in-1 Plot for FFT with IDAPro and Optimization Off, Levels 1-5

```
The regression equation is
LU (Execution Time) = - 417 + 290 Level

Predictor      Coef  SE Coef      T      P
Constant    -416.89    72.50  -5.75  0.000
Level        290.39    21.86  13.28  0.000

S = 154.576   R-Sq = 88.5%   R-Sq(adj) = 88.0%

Analysis of Variance

Source          DF       SS       MS       F      P
Regression       1  4216262  4216262  176.46  0.000
Residual Error  23   549560    23894
Total           24  4765822

Unusual Observations

                       LU
            (Execution
Obs  Level       Time)     Fit  SE Fit  Residual  St Resid
 21   5.00      1342.9  1035.0    53.5     307.9     2.12R

R denotes an observation with a large standardized residual.
```

Figure A.64. Regression Model for LU w/ IDAPro and Optimization Off, Levels 1-5

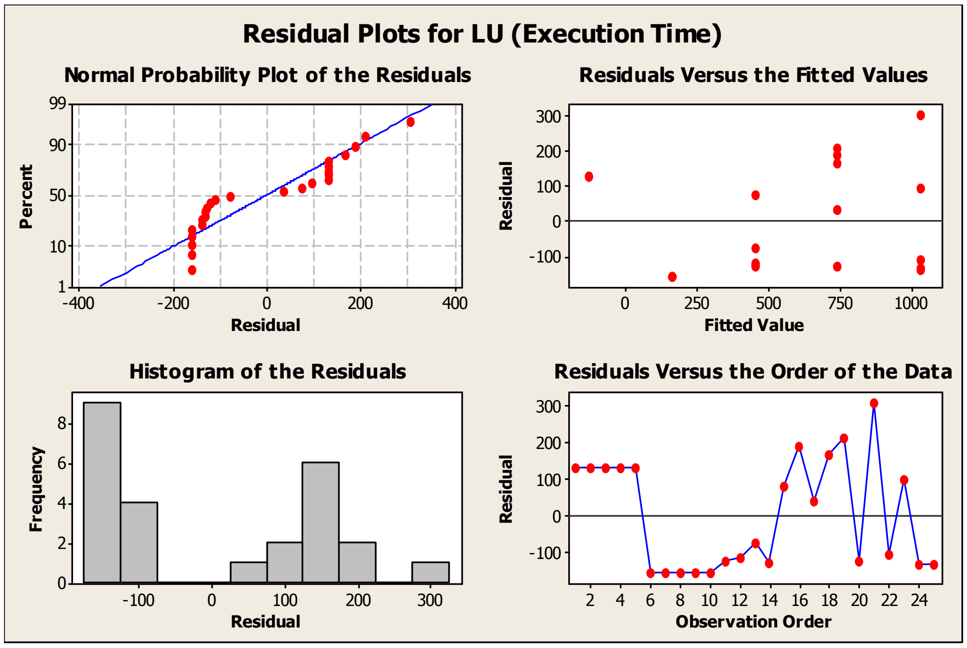


Figure A.65. 4-in-1 Plot for LU with IDAPro and Optimization Off, Levels 1-5

```
The regression equation is
Monte (Execution Time) = - 1805 + 1289 Level

Predictor      Coef  SE Coef      T      P
Constant    -1804.9    318.8  -5.66  0.000
Level       1288.87    96.13  13.41  0.000

S = 679.708   R-Sq = 88.7%   R-Sq(adj) = 88.2%

Analysis of Variance

Source          DF        SS        MS       F      P
Regression       1  83059113  83059113  179.78  0.000
Residual Error  23  10626056    462002
Total           24  93685169
```

Figure A.66. Regression Model for Monte w/ IDAPro and Optimization Off, Levels 1-5

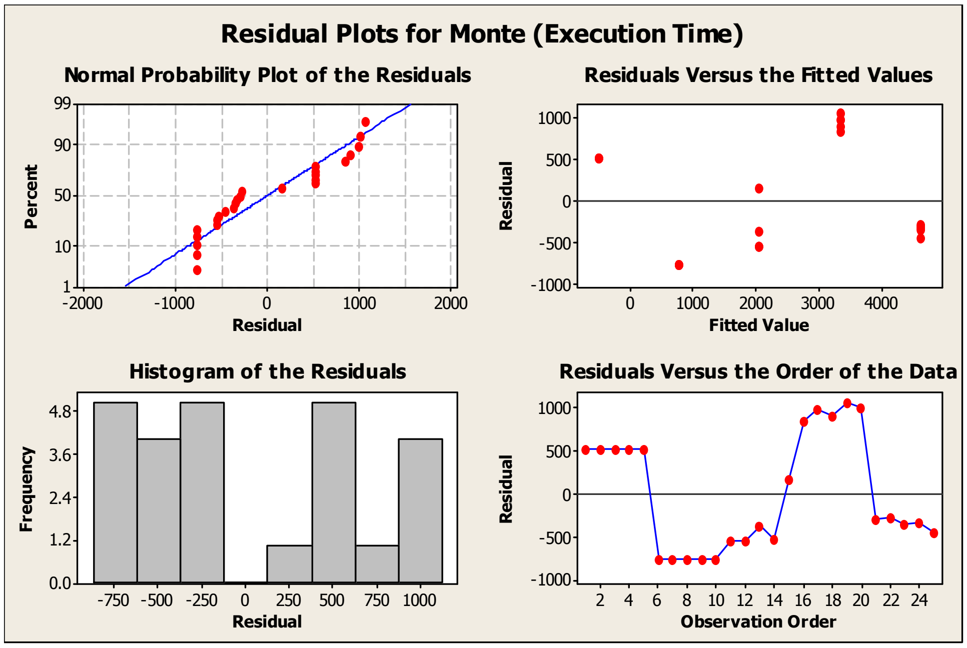


Figure A.67. 4-in-1 Plot for Monte with IDAPro and Optimization Off, Levels 1-5

```
The regression equation is
SOR (Execution Time) = - 10091 + 7805 Level

Predictor     Coef  SE Coef      T      P
Constant    -10091     1833  -5.50  0.000
Level       7805.5    552.8  14.12  0.000

S = 3908.69   R-Sq = 89.7%   R-Sq(adj) = 89.2%

Analysis of Variance

Source          DF          SS          MS       F      P
Regression       1  3046285658  3046285658  199.39  0.000
Residual Error  23   351390457    15277846
Total           24  3397676115
```

Figure A.68. Regression Model for SOR w/ IDAPro and Optimization Off, Levels 1-5

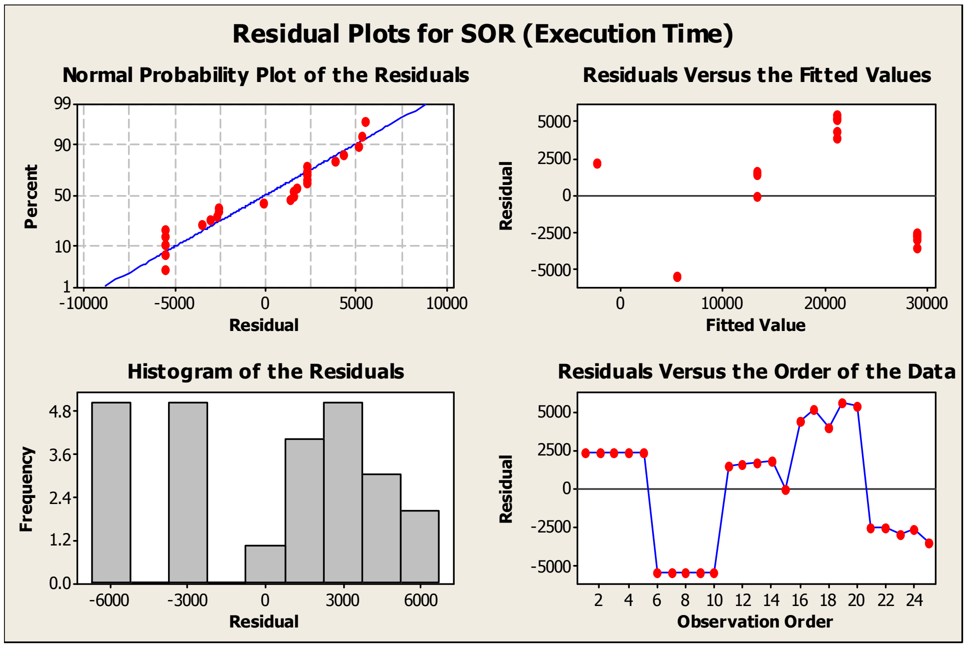


Figure A.69. 4-in-1 Plot for SOR with IDAPro and Optimization Off, Levels 1-5

```
The regression equation is
Sparse (Execution Time) = - 27455 + 19249 Level

Predictor     Coef   SE Coef       T      P
Constant    -27455      4231   -6.49  0.000
Level        19249      1276   15.09  0.000

S = 9019.99   R-Sq = 90.8%   R-Sq(adj) = 90.4%

Analysis of Variance

Source           DF           SS           MS       F      P
Regression        1  18526101803  18526101803  227.70  0.000
Residual Error   23   1871283308     81360144
Total            24  20397385111
```

Figure A.70. Regression Model for Sparse w/ IDAPro and Optimization Off, Levels 1-5

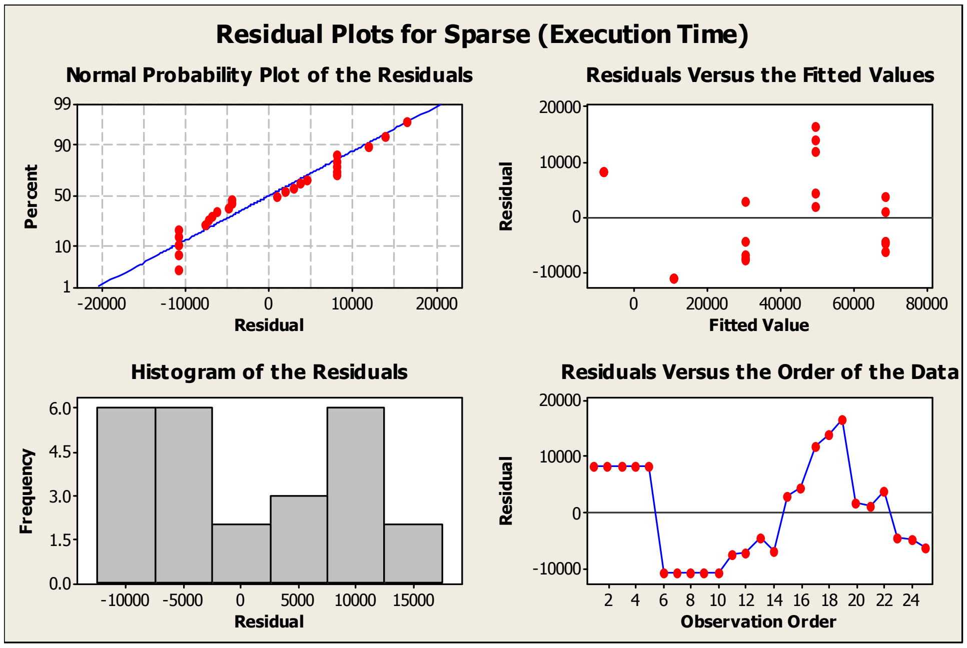


Figure A.71. 4-in-1 Plot for Sparse with IDAPro and Optimization Off, Levels 1-5

Table A.6. Mean of Differences (Is there a statistically significant difference present?) using FFT with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	YES	YES
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.7. Mean of Differences (Is there a statistically significant difference present?) using LU with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	YES	YES
4 (hidden function w/ 8 threads)	X	X	X	X	YES
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.8. Mean of Differences (Is there a statistically significant difference present?) using Monte with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	YES	YES
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.9. Mean of Differences (Is there a statistically significant difference present?) using SOR with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	YES	YES
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.10. Mean of Differences (Is there a statistically significant difference present?) using Sparse with Optimization Off and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	YES	YES
4 (hidden function w/ 8 threads)	X	X	X	X	YES
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.11. Ggrep with Optimization On and IDAPro, Levels 1-5

Level	Ggrep Expression	Mean	St-Dev	90% Confidence Interval
1 (baseline)	Expression 1	16.159	0.112	[16.052, 16.265]
2 (hidden function in use)	Expression 1	16.096	0.011	[16.086, 16.107]
3 (hidden function w/ 4 threads)	Expression 1	16.115	0.026	[16.090, 16.139]
4 (hidden function w/ 8 threads)	Expression 1	16.109	0.007	[16.102, 16.115]
5 (hidden function w/ 12 threads)	Expression 1	16.217	0.151	[16.073, 16.361]
1 (baseline)	Expression 2	4.7282	0.0138	[4.7150, 4.7414]
2 (hidden function in use)	Expression 2	4.744	0.0082	[4.7362, 4.7518]
3 (hidden function w/ 4 threads)	Expression 2	4.753	0.0067	[4.7466, 4.7594]
4 (hidden function w/ 8 threads)	Expression 2	4.75	0	[4.750, 4.750]
5 (hidden function w/ 12 threads)	Expression 2	4.7532	0.0072	[4.7464, 4.76]
1 (baseline)	Expression 3	193.35	0.23	[193.13, 193.57]
2 (hidden function in use)	Expression 3	194.46	0.04	[194.42, 194.50]
3 (hidden function w/ 4 threads)	Expression 3	194.59	0.09	[194.51, 194.67]
4 (hidden function w/ 8 threads)	Expression 3	194.84	0.37	[194.50, 195.19]
5 (hidden function w/ 12 threads)	Expression 3	196.1	3.56	[192.71, 199.5]

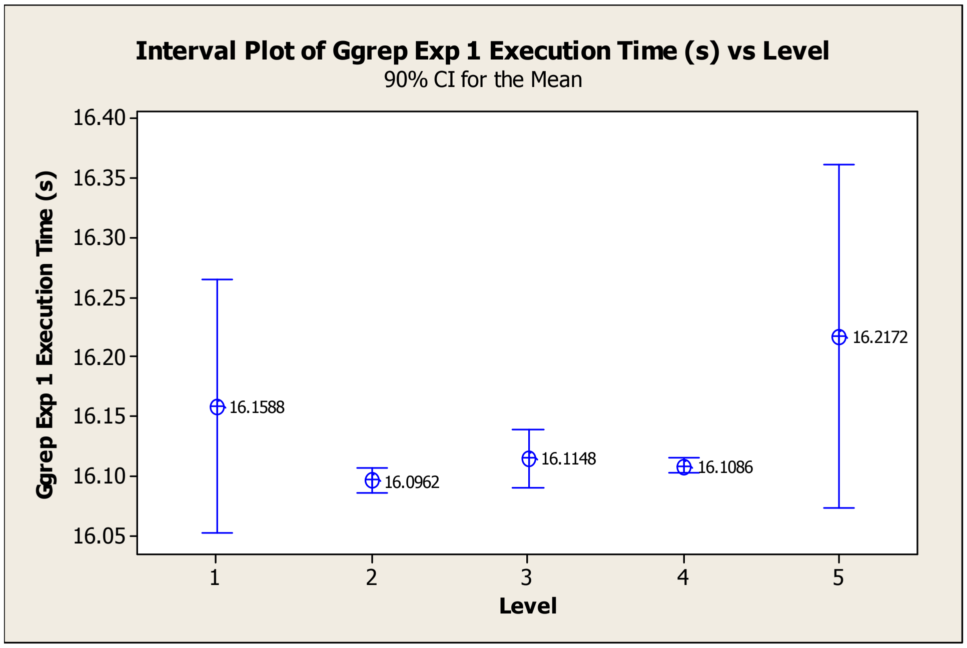


Figure A.72. Mean Interval Plot of Ggrep Expression 1 w/ IDAPro and Optimization On

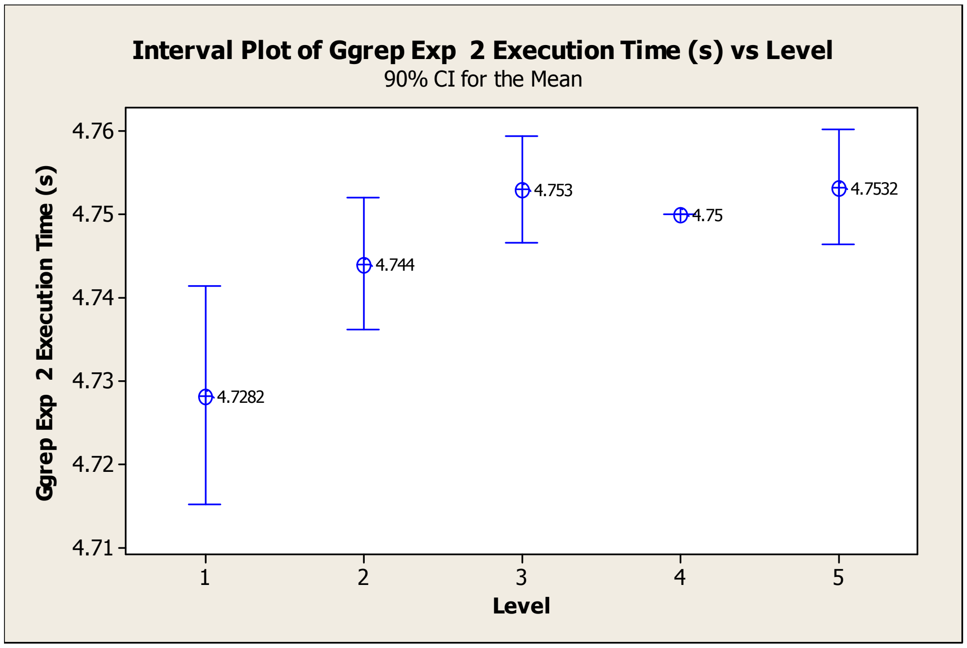


Figure A.73. Mean Interval Plot of Ggrep Expression 2 w/ IDAPro and Optimization On

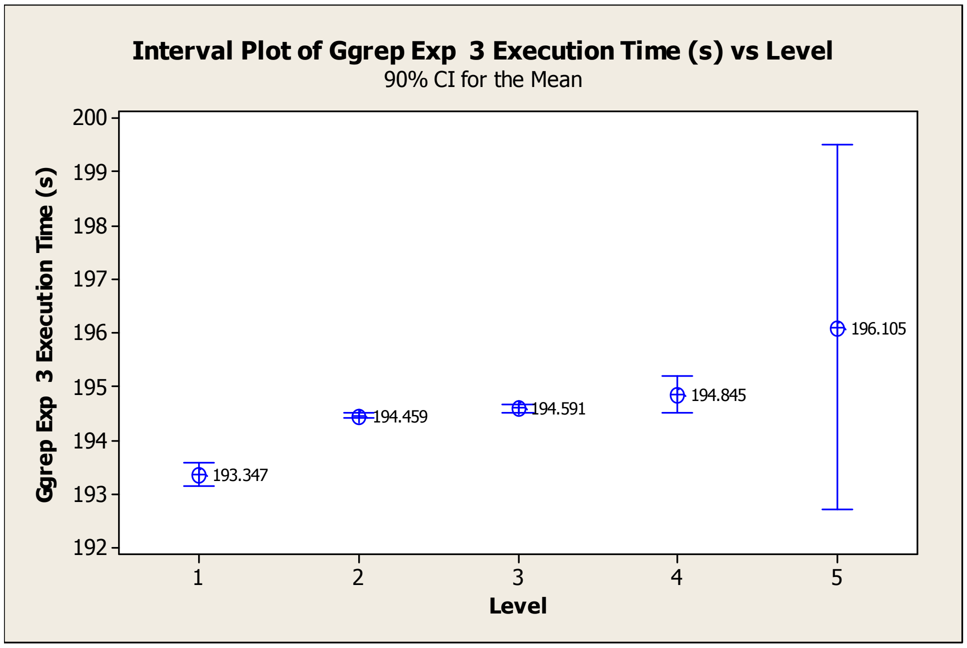


Figure A.74. Mean Interval Plot of Ggrep Expression 3 w/ IDAPro and Optimization On

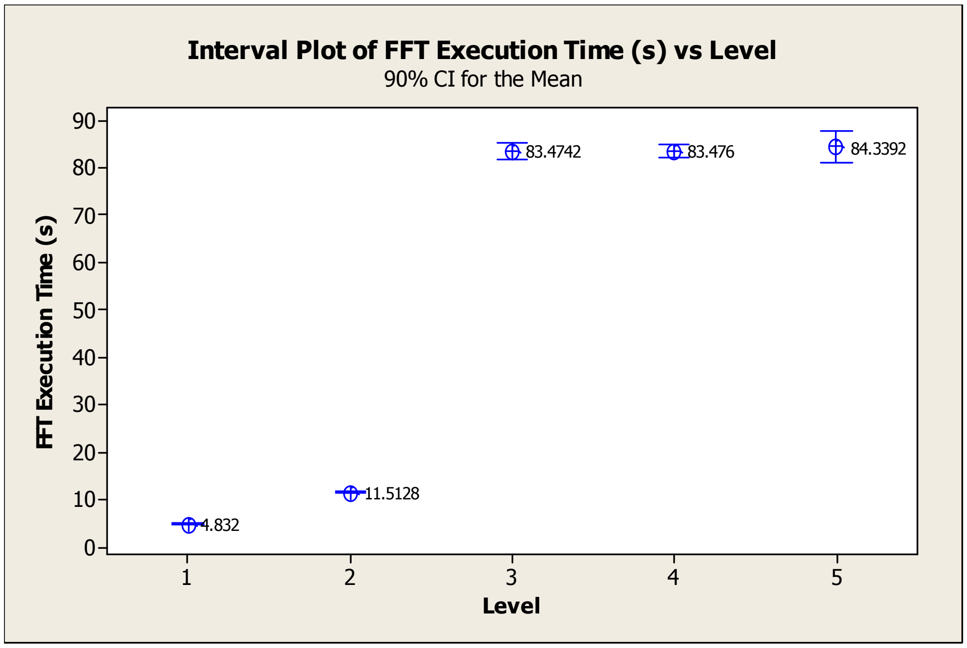


Figure A.75. Mean Interval Plot of FFT w/ IDAPro and Optimization On

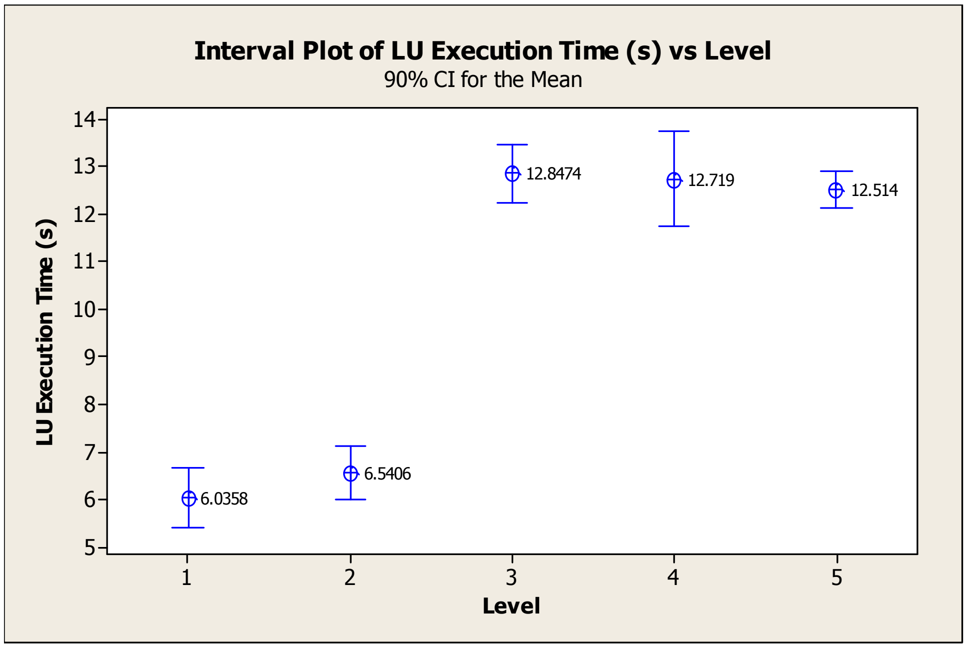


Figure A.76. Mean Interval Plot of LU w/ IDAPro and Optimization On

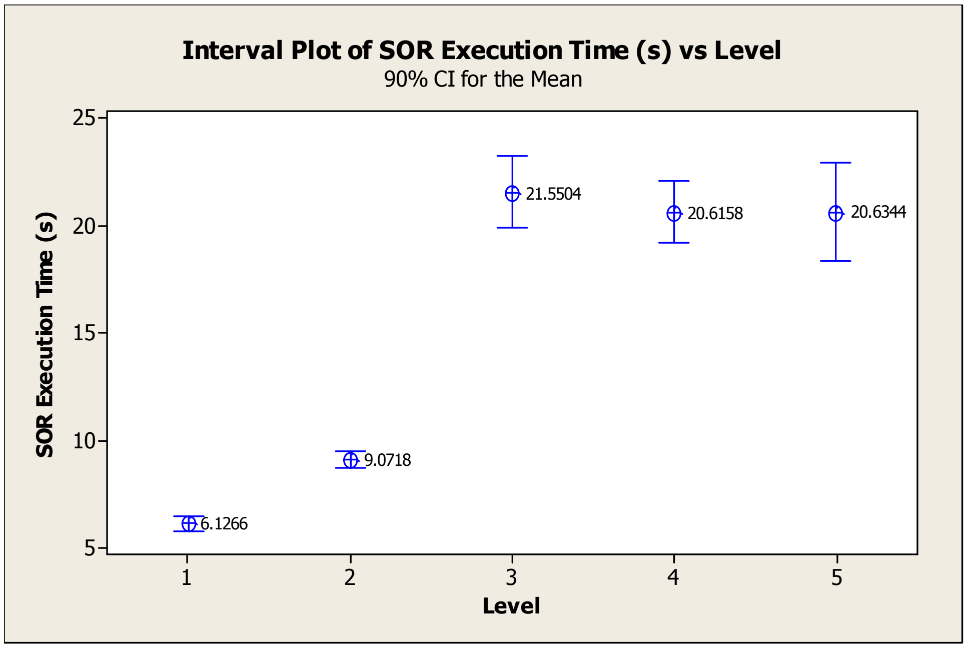


Figure A.77. Mean Interval Plot of SOR w/ IDAPro and Optimization On

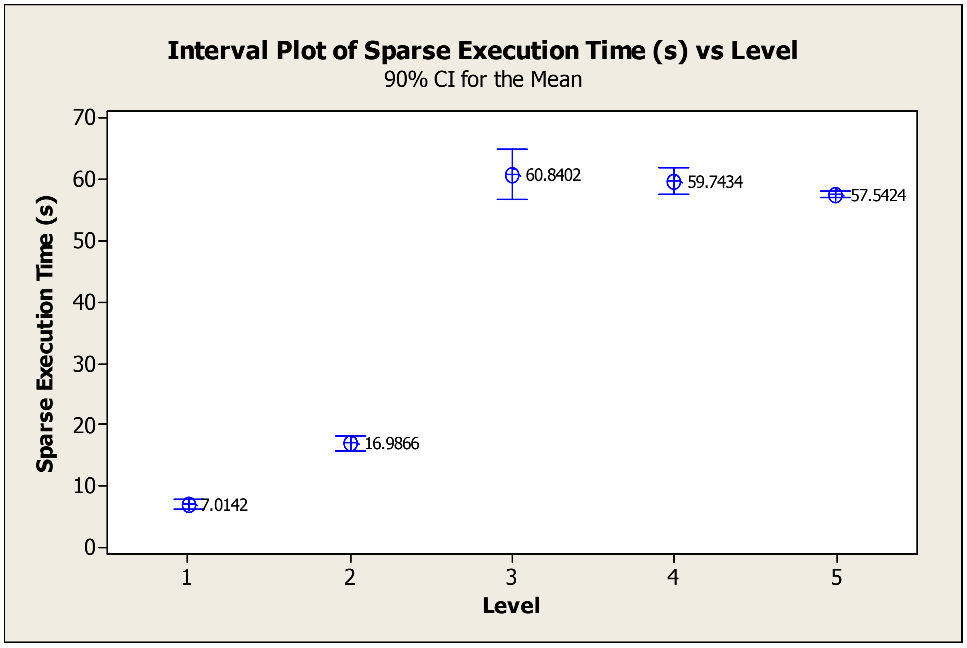


Figure A.78. Mean Interval Plot of Sparse w/ IDAPro and Optimization On

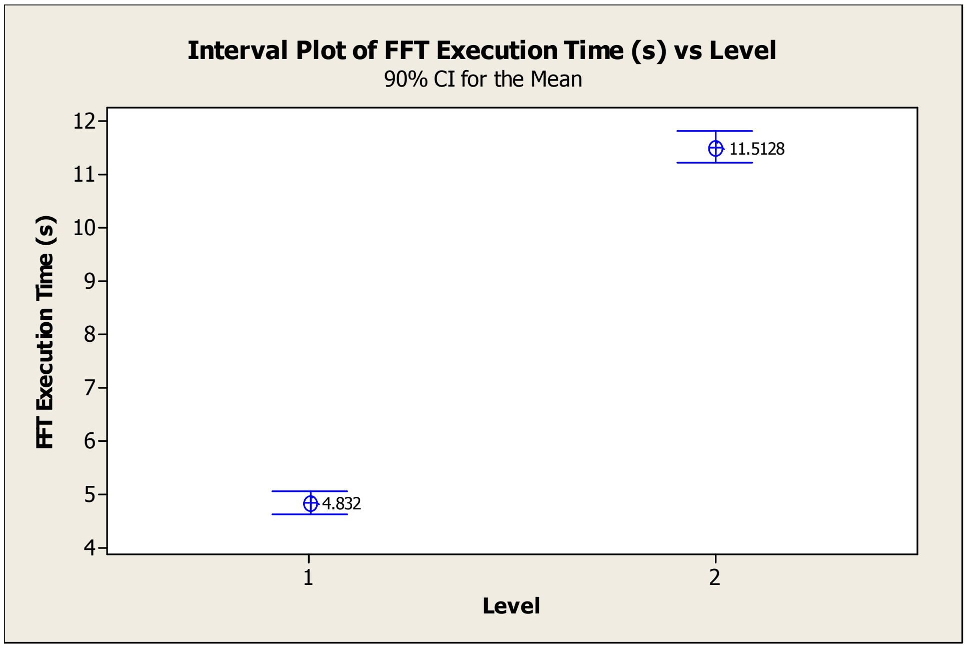


Figure A.79. Mean Interval Plot of FFT w/ IDAPro and Optimization On, Levels 1 and 2

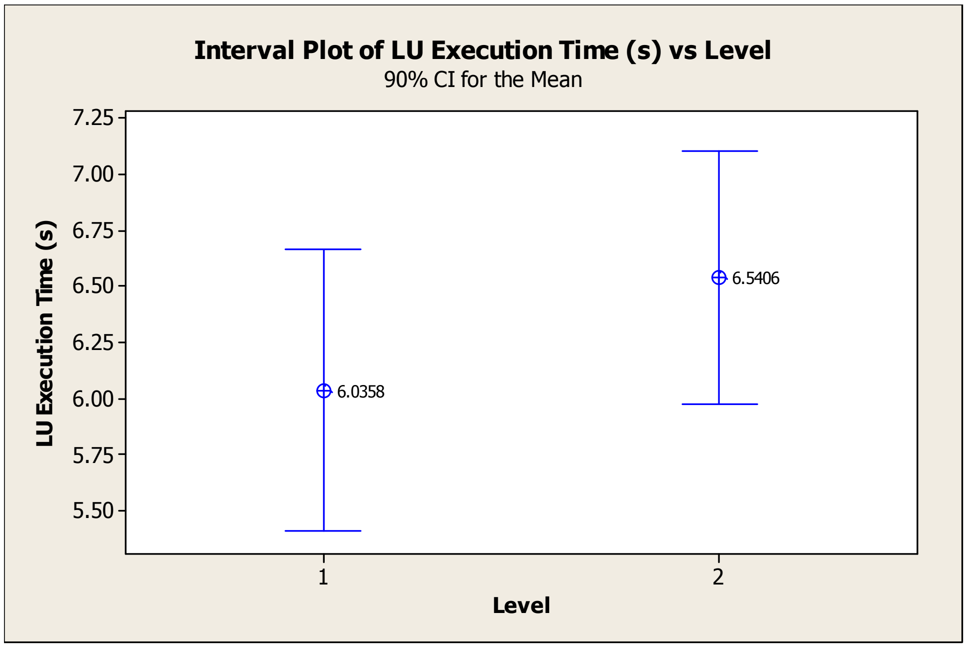


Figure A.80. Mean Interval Plot of LU w/ IDAPro and Optimization On, Levels 1 and 2

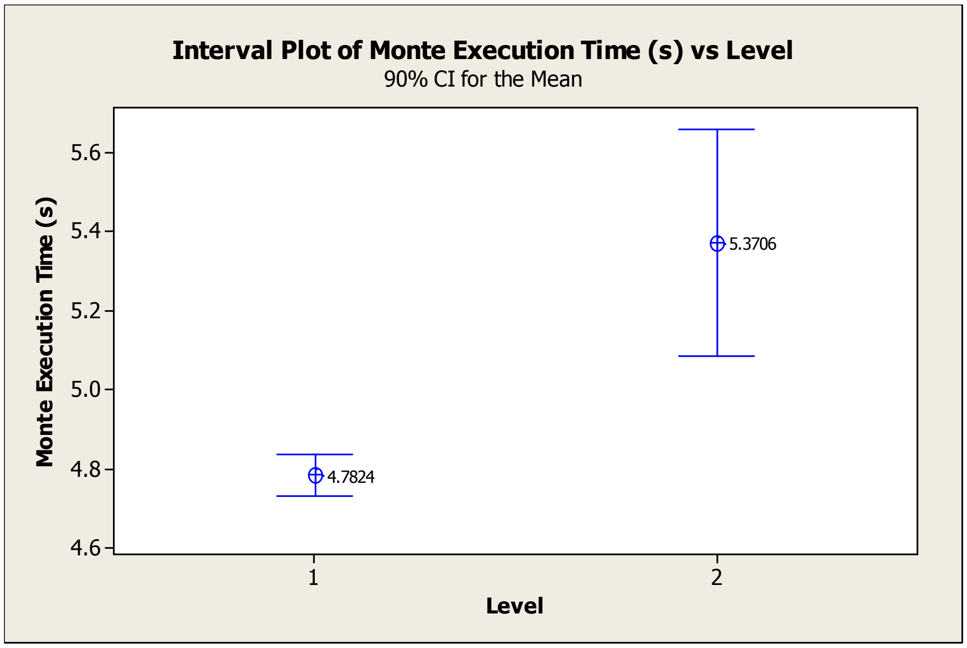


Figure A.81. Mean Interval Plot of Monte w/ IDAPro and Optimization On, Levels 1 and 2

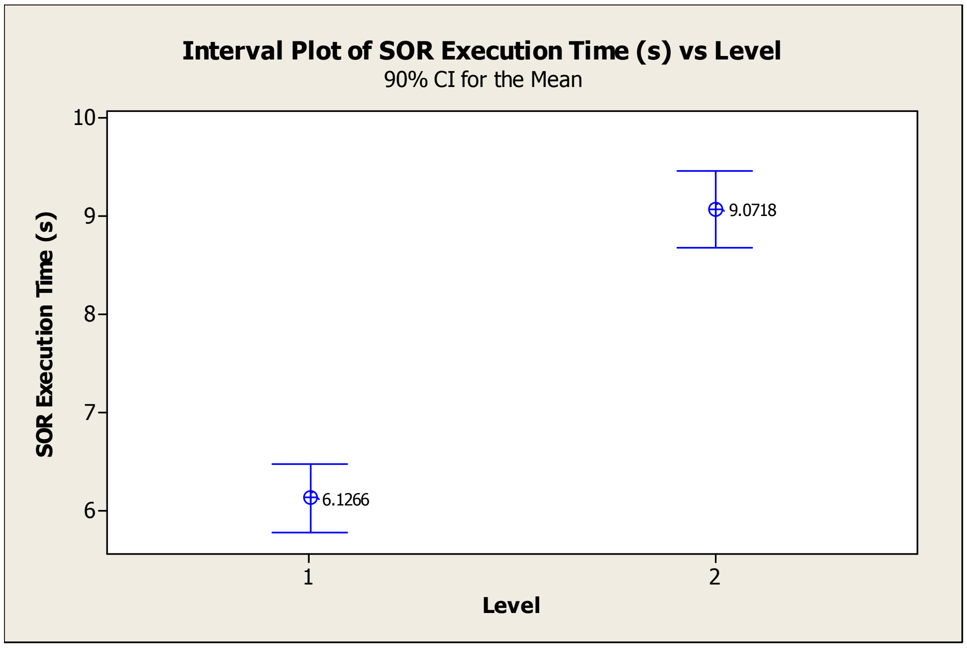


Figure A.82. Mean Interval Plot of SOR w/ IDAPro and Optimization On, Levels 1 and 2

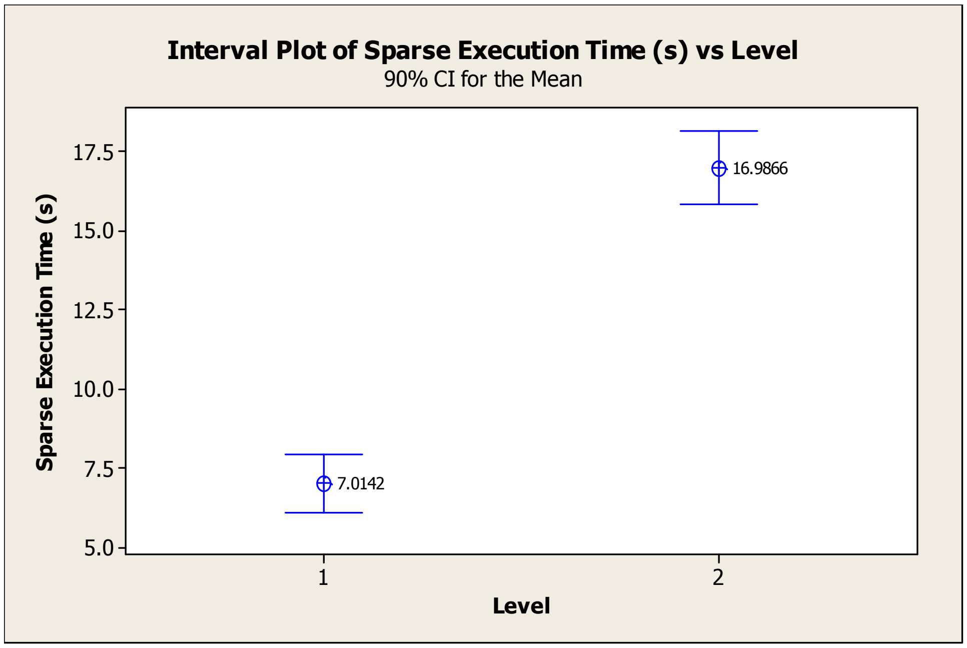


Figure A.83. Mean Interval Plot of Sparse w/ IDAPro and Optimization On, Levels 1 and 2

```
The regression equation is
Ggrep Exp 1 (Execution Time) = 16.1 + 0.0129 Level

Predictor      Coef   SE Coef       T      P
Constant    16.1004    0.0421  382.51  0.000
Level       0.01292   0.01269    1.02  0.319

S = 0.0897393   R-Sq = 4.3%   R-Sq(adj) = 0.2%

Analysis of Variance

Source          DF        SS        MS     F      P
Regression       1  0.008346  0.008346  1.04  0.319
Residual Error  23  0.185222  0.008053
Total           24  0.193569

Unusual Observations
             Ggrep Exp
                     1
            (Execution
Obs  Level       Time)      Fit  SE Fit  Residual  St Resid
  2   1.00     16.3310  16.1133  0.0311    0.2177     2.59R
 22   5.00     16.4560  16.1650  0.0311    0.2910     3.46R

R denotes an observation with a large standardized residual.
```

Figure A.84. Regression Model for Ggrep Expression 1 w/ IDAPro and Optimization On, Levels 1-5

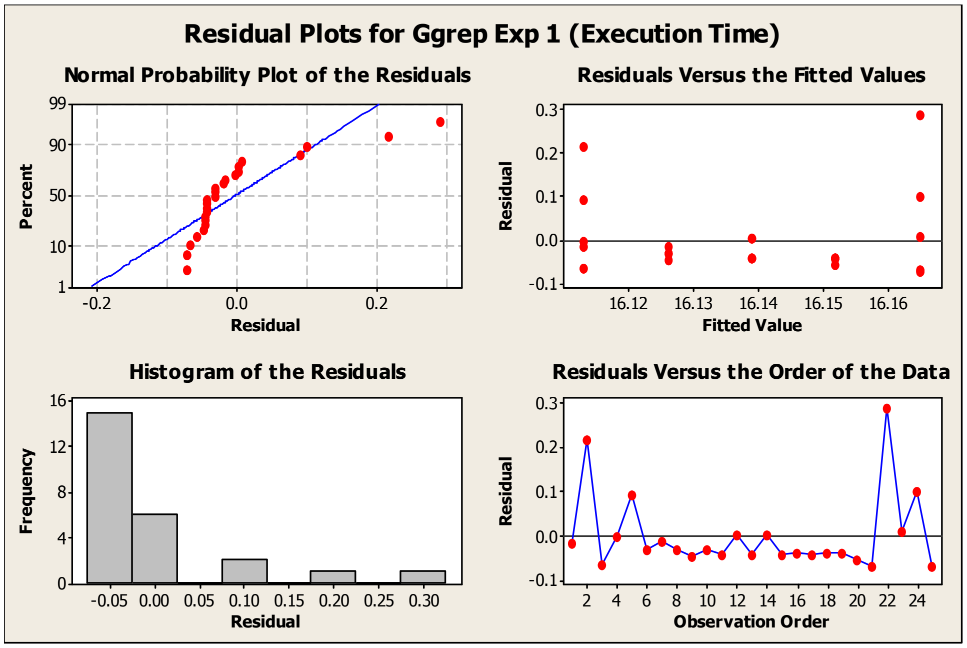


Figure A.85. 4-in-1 Plot for Ggrep Expression 1 with IDAPro and Optimization On, Levels 1-5

```
The regression equation is
Ggrep Exp  2 (Execution Time) = 4.73 + 0.00560 Level

Predictor       Coef    SE Coef        T      P
Constant     4.72888    0.00441  1071.72  0.000
Level       0.005600   0.001330     4.21  0.000

S = 0.00940731   R-Sq = 43.5%   R-Sq(adj) = 41.1%

Analysis of Variance

Source          DF         SS         MS      F      P
Regression       1  0.0015680  0.0015680  17.72  0.000
Residual Error  23  0.0020354  0.0000885
Total           24  0.0036034

Unusual Observations

             Ggrep Exp
                     2
            (Execution
Obs  Level       Time)      Fit    SE Fit  Residual  St Resid
 11   3.00     4.76500  4.74568  0.00188   0.01932      2.10R

R denotes an observation with a large standardized residual.
```

Figure A.86. Regression Model for Ggrep Expression 2 w/ IDAPro and Optimization On, Levels 1-5

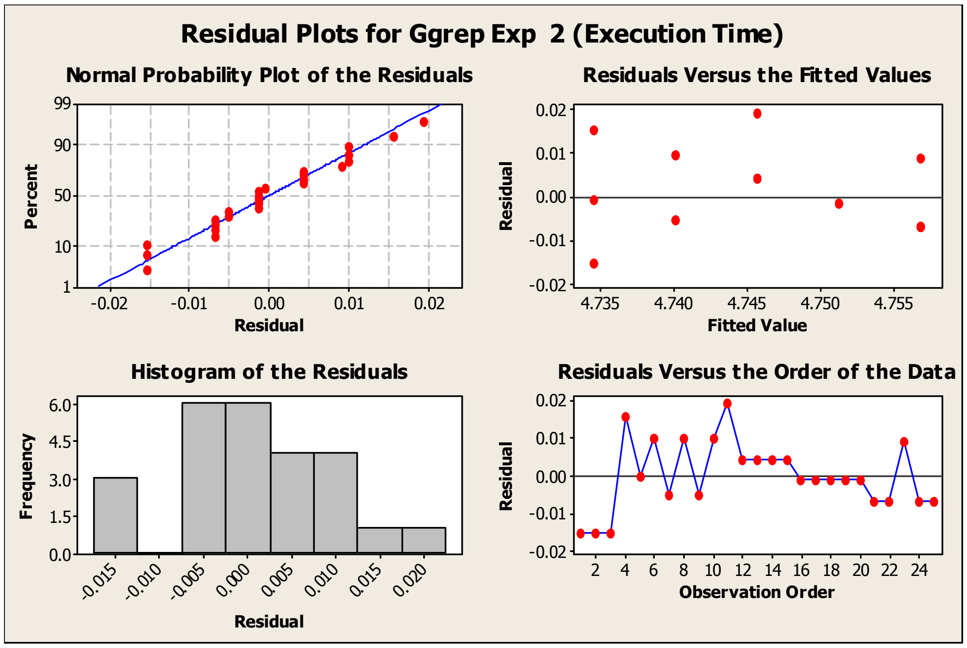


Figure A.87. 4-in-1 Plot for Ggrep Expression 2 with IDAPro and Optimization On, Levels 1-5

```
The regression equation is
Ggrep Exp  3 (Execution Time) = 193 + 0.590 Level

Predictor      Coef  SE Coef       T      P
Constant    192.899    0.716  269.32  0.000
Level        0.5901   0.2160    2.73  0.012

S = 1.52702   R-Sq = 24.5%   R-Sq(adj) = 21.2%

Analysis of Variance

Source          DF      SS      MS     F      P
Regression       1  17.409  17.409  7.47  0.012
Residual Error  23  53.631   2.332
Total           24  71.040

Unusual Observations

             Ggrep Exp
                     3
             (Execution
Obs  Level        Time)      Fit  SE Fit  Residual  St Resid
 21   5.00      202.469  195.849   0.529     6.620     4.62R

R denotes an observation with a large standardized residual.
```

Figure A.88. Regression Model for Ggrep Expression 3 w/ IDAPro and Optimization On, Levels 1-5

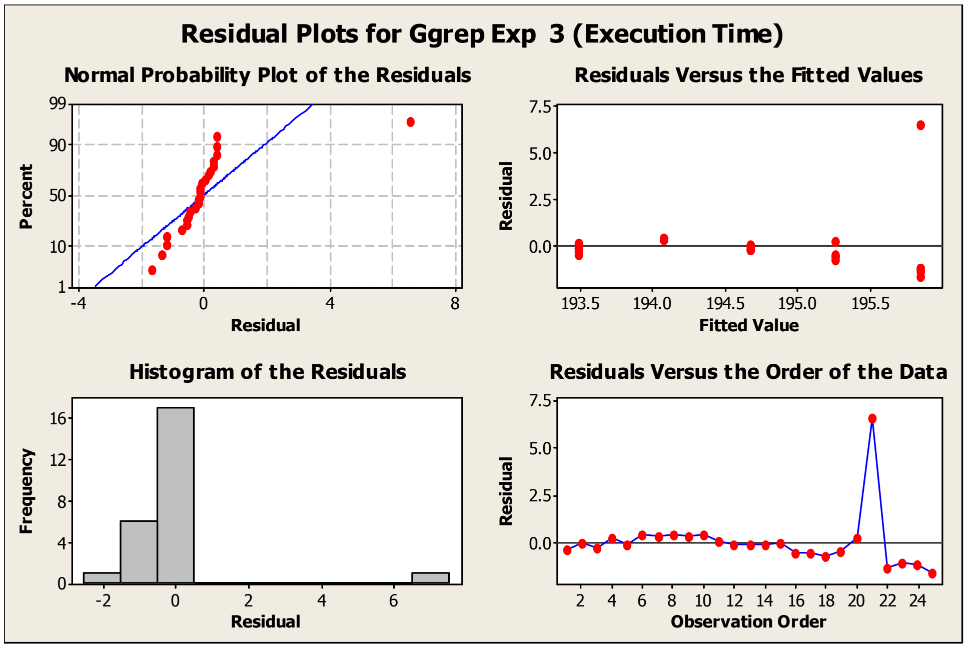


Figure A.89. 4-in-1 Plot for Ggrep Expression 3 with IDAPro and Optimization On, Levels 1-5

Table A.12. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 1 with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	NO	NO	NO
2 (hidden function in use)	X	X	NO	NO	NO
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.13. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 2 with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	YES	YES	YES
2 (hidden function in use)	X	X	YES	NO	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.14. Mean of Differences (Is there a statistically significant difference present?) using Ggrep Expression 3 with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	NO
2 (hidden function in use)	X	X	YES	YES	NO
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.15. SciMark2 with Optimization On and IDAPro, Levels 1-5

Level	Function	Mean	St-Dev	90% Confidence Interval
1 (baseline)	FFT	4.832	0.2364	[4.6066, 5.0574]
2 (hidden function in use)	FFT	11.513	0.323	[11.205, 11.821]
3 (hidden function w/ 4 threads)	FFT	83.474	1.853	[81.708, 85.241]
4 (hidden function w/ 8 threads)	FFT	83.476	1.405	[82.136, 84.816]
5 (hidden function w/ 12 threads)	FFT	84.339	3.423	[81.076, 87.602]
1 (baseline)	LU	6.0358	0.6595	[5.4070, 6.6646]
2 (hidden function in use)	LU	6.5406	0.5956	[5.9728, 71.084]
3 (hidden function w/ 4 threads)	LU	12.847	0.646	[12.231, 13.464]
4 (hidden function w/ 8 threads)	LU	12.719	1.052	[11.716, 13.722]
5 (hidden function w/ 12 threads)	LU	12.514	0.408	[12.125, 12.903]
1 (baseline)	Monte	4.7824	0.0539	[4.7310, 4.8338]
2 (hidden function in use)	Monte	5.3706	0.2997	[5.0849, 5.6563]
3 (hidden function w/ 4 threads)	Monte	N/A	N/A	N/A
4 (hidden function w/ 8 threads)	Monte	N/A	N/A	N/A
5 (hidden function w/ 12 threads)	Monte	N/A	N/A	N/A
1 (baseline)	SOR	6.1266	0.3651	[5.7785, 6.4747]
2 (hidden function in use)	SOR	9.0718	0.4013	[8.6892, 9.4544]
3 (hidden function w/ 4 threads)	SOR	21.55	1.762	[19.871, 23.230]
4 (hidden function w/ 8 threads)	SOR	20.616	1.518	[19.168, 22.063]
5 (hidden function w/ 12 threads)	SOR	20.634	2.437	[18.311, 22.958]
1 (baseline)	Sparse	7.0142	0.9644	[6.0947, 7.9337]
2 (hidden function in use)	Sparse	16.987	1.222	[15.822, 18.151]
3 (hidden function w/ 4 threads)	Sparse	60.84	4.283	[56.757, 64.923]
4 (hidden function w/ 8 threads)	Sparse	59.743	2.2	[57.646, 61.841]
5 (hidden function w/ 12 threads)	Sparse	57.542	0.592	[56.978, 58.107]

```
The regression equation is
FFT (Execution Time) = - 15.8 + 23.1 Level

Predictor      Coef  SE Coef      T      P
Constant    -15.766    8.633  -1.83  0.081
Level        23.098    2.603   8.87  0.000

S = 18.4056   R-Sq = 77.4%   R-Sq(adj) = 76.4%

Analysis of Variance

Source          DF     SS     MS      F      P
Regression       1  26675  26675  78.74  0.000
Residual Error  23   7792    339
Total           24  34467
```

Figure A.90. Regression Model for FFT w/ IDAPro and Optimization On, Levels 1-5

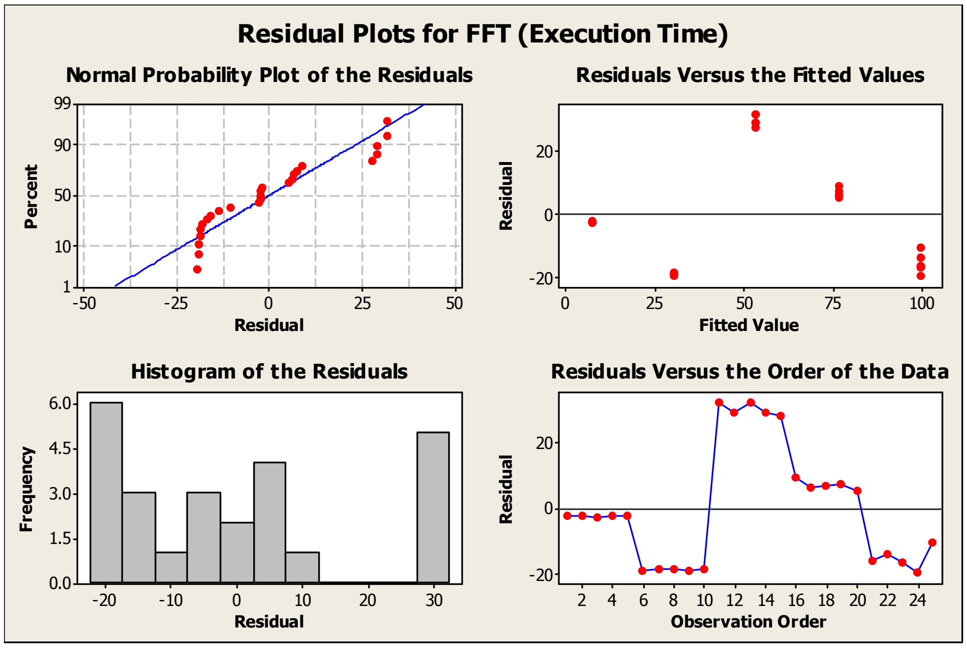


Figure A.91. 4-in-1 Plot for FFT with IDAPro and Optimization On, Levels 1-5

```
The regression equation is
LU (Execution Time) = 4.39 + 1.91 Level

Predictor     Coef  SE Coef     T      P
Constant    4.3909   0.8410  5.22  0.000
Level       1.9135   0.2536  7.55  0.000

S = 1.79293   R-Sq = 71.2%   R-Sq(adj) = 70.0%

Analysis of Variance

Source          DF      SS      MS      F      P
Regression       1  183.07  183.07  56.95  0.000
Residual Error  23   73.94    3.21
Total           24  257.01

Unusual Observations

                       LU
            (Execution
Obs  Level       Time)     Fit  SE Fit  Residual  St Resid
 12   3.00      13.837  10.131   0.359     3.706     2.11R

R denotes an observation with a large standardized residual.
```

Figure A.92. Regression Model for LU w/ IDAPro and Optimization On, Levels 1-5

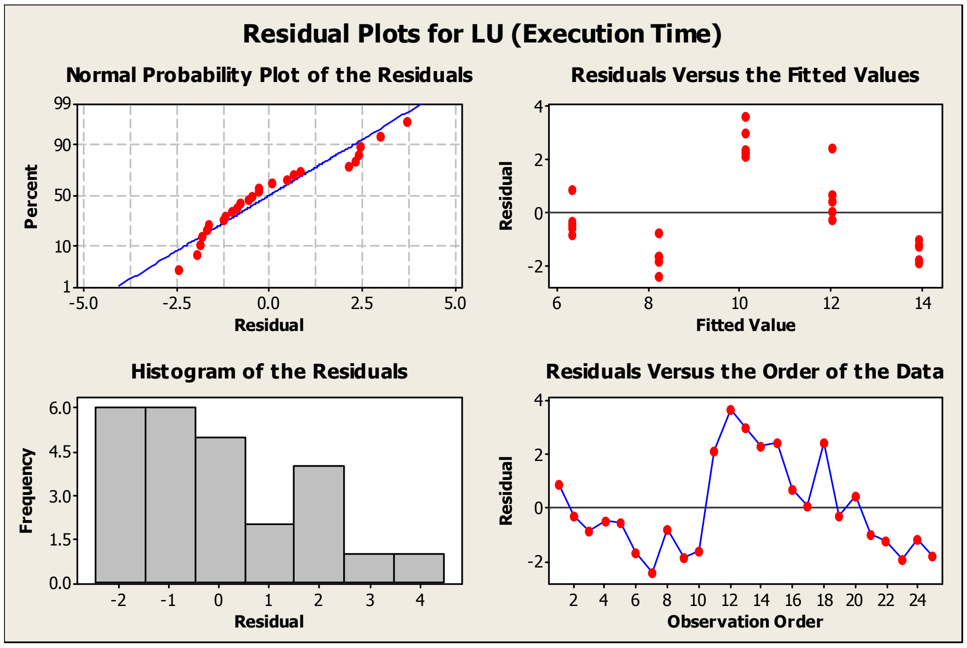


Figure A.93. 4-in-1 Plot for LU with IDAPro and Optimization On, Levels 1-5

```
The regression equation is
Monte (Execution Time) = 4.19 + 0.588 Level

Predictor     Coef  SE Coef      T      P
Constant    4.1942   0.2153  19.48  0.000
Level       0.5882   0.1362   4.32  0.003

S = 0.215323   R-Sq = 70.0%   R-Sq(adj) = 66.2%

Analysis of Variance

Source          DF       SS       MS      F      P
Regression       1  0.86495  0.86495  18.66  0.003
Residual Error   8  0.37091  0.04636
Total            9  1.23586

Unusual Observations

                 Monte
            (Execution
Obs  Level       Time)     Fit  SE Fit  Residual  St Resid
  8   2.00      5.8140  5.3706  0.0963    0.4434     2.30R

R denotes an observation with a large standardized residual.
```

Figure A.94. Regression Model for Monte w/ IDAPro and Optimization On, Levels 1-5

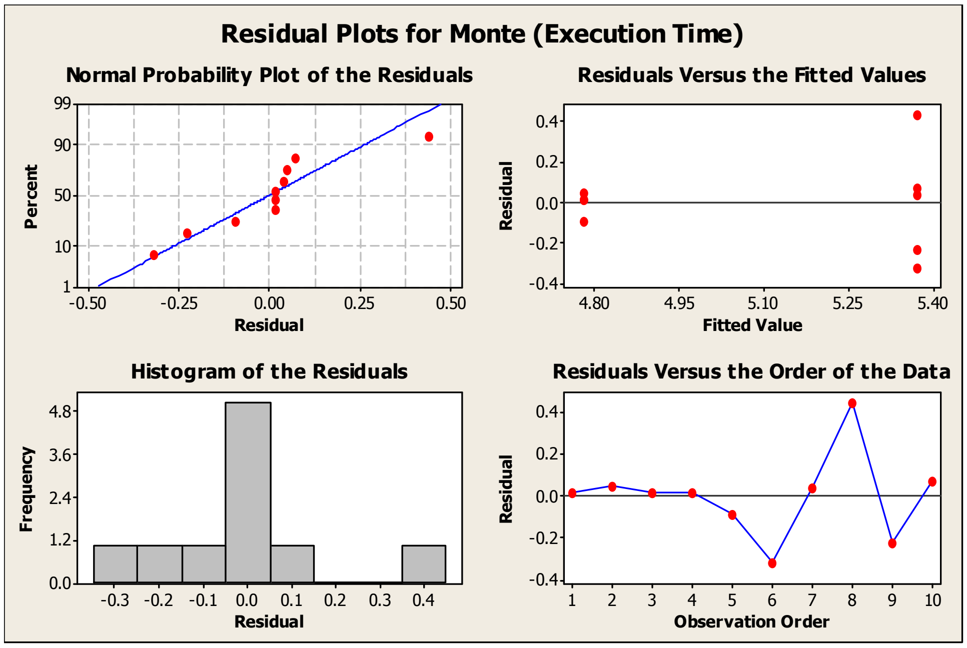


Figure A.95. 4-in-1 Plot for Monte with IDAPro and Optimization On, Levels 1-5

```
The regression equation is
SOR Execution Time (s) = 3.43 + 4.06 Level

Predictor     Coef  SE Coef     T      P
Constant     3.432    1.737  1.98  0.060
Level       4.0560   0.5237  7.74  0.000

S = 3.70313   R-Sq = 72.3%   R-Sq(adj) = 71.1%

Analysis of Variance

Source          DF       SS      MS      F      P
Regression       1   822.54  822.54  59.98  0.000
Residual Error  23   315.40   13.71
Total           24  1137.94

Unusual Observations

                    SOR
            Execution
Obs  Level   Time (s)     Fit  SE Fit  Residual  St Resid
 13   3.00     23.545  15.600   0.741     7.945     2.19R

R denotes an observation with a large standardized residual.
```

Figure A.96. Regression Model for SOR w/ IDAPro and Optimization On, Levels 1-5

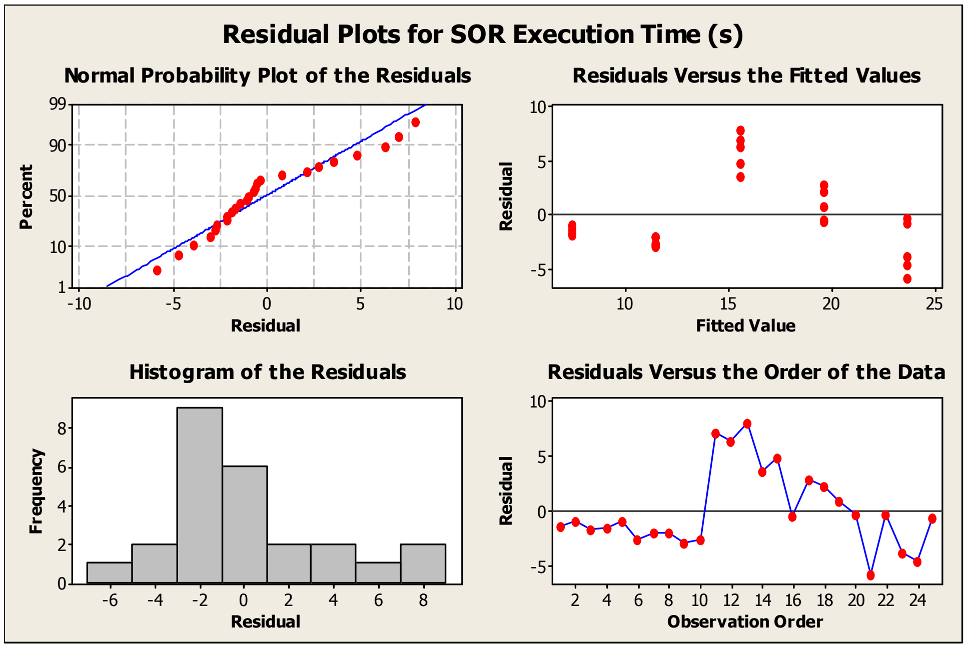


Figure A.97. 4-in-1 Plot for SOR with IDAPro and Optimization On, Levels 1-5

```
The regression equation is
Sparse (Execution Time) = - 2.72 + 14.4 Level

Predictor     Coef  SE Coef      T      P
Constant    -2.719    5.791  -0.47  0.643
Level       14.381    1.746   8.24  0.000

S = 12.3462   R-Sq = 74.7%   R-Sq(adj) = 73.6%

Analysis of Variance

Source          DF     SS     MS      F      P
Regression       1  10341  10341  67.84  0.000
Residual Error  23   3506    152
Total           24  13847

Unusual Observations

                Sparse
              (Execution
Obs  Level        Time)     Fit  SE Fit  Residual  St Resid
 13   3.00        67.36   40.43    2.47     26.93     2.23R

R denotes an observation with a large standardized residual.
```

Figure A.98. Regression Model for Sparse w/ IDAPro and Optimization On, Levels 1-5

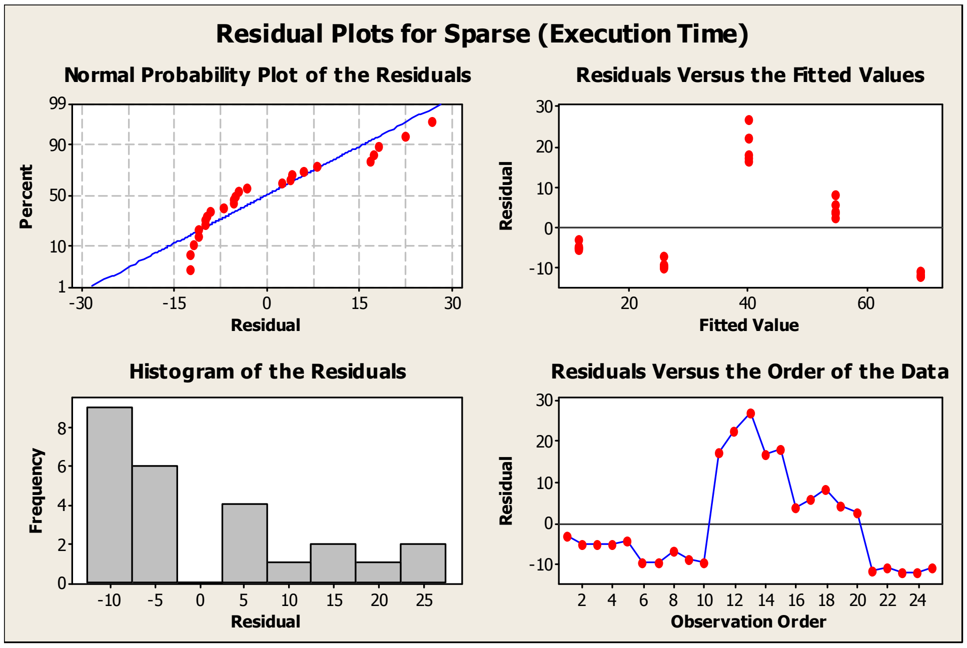


Figure A.99. 4-in-1 Plot for Sparse with IDAPro and Optimization On, Levels 1-5

Table A.16. Mean of Differences (Is there a statistically significant difference present?) using FFT with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.17. Mean of Differences (Is there a statistically significant difference present?) using LU with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	NO	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.18. Mean of Differences (Is there a statistically significant difference present?) using Monte with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	N/A	N/A	N/A
2 (hidden function in use)	X	X	N/A	N/A	N/A
3 (hidden function w/ 4 threads)	X	X	X	N/A	N/A
4 (hidden function w/ 8 threads)	X	X	X	X	N/A
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.19. Mean of Differences (Is there a statistically significant difference present?) using SOR with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	NO	NO
4 (hidden function w/ 8 threads)	X	X	X	X	NO
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.20. Mean of Differences (Is there a statistically significant difference present?) using Sparse with Optimization On and IDAPro

Level	Level 1	Level 2	Level 3	Level 4	Level 5
1 (baseline)	X	YES	YES	YES	YES
2 (hidden function in use)	X	X	YES	YES	YES
3 (hidden function w/ 4 threads)	X	X	X	YES	NO
4 (hidden function w/ 8 threads)	X	X	X	X	YES
5 (hidden function w/ 12 threads)	X	X	X	X	X

Table A.21. Mean of Differences for Ggrep Expressions 1-3 with IDAPro, Optimization Off vs Optimization On (Is there a statistically significant difference between the same levels when optimization is on and off?)

	Optimization-Off versus Optimization-On				
	Level 1	Level 2	Level 3	Level 4	Level 5
Ggrep Expression 1	Yes	Yes	Yes	Yes	Yes
Ggrep Expression 2	Yes	Yes	Yes	Yes	Yes
Ggrep Expression 3	Yes	Yes	Yes	Yes	Yes

Table A.22. Mean of Differences for SciMark2 with IDAPro, Optimization Off vs Optimization On (Is there a statistically significant difference between the same levels when optimization is on and off?)

	Optimization-Off versus Optimization-On				
Function	Level 1	Level 2	Level 3	Level 4	Level 5
FFT	Yes	Yes	Yes	Yes	Yes
LU	Yes	Yes	Yes	Yes	Yes
Monte	Yes	Yes	N/A	N/A	N/A
SOR	Yes	Yes	Yes	Yes	Yes
Sparse	Yes	Yes	Yes	Yes	Yes

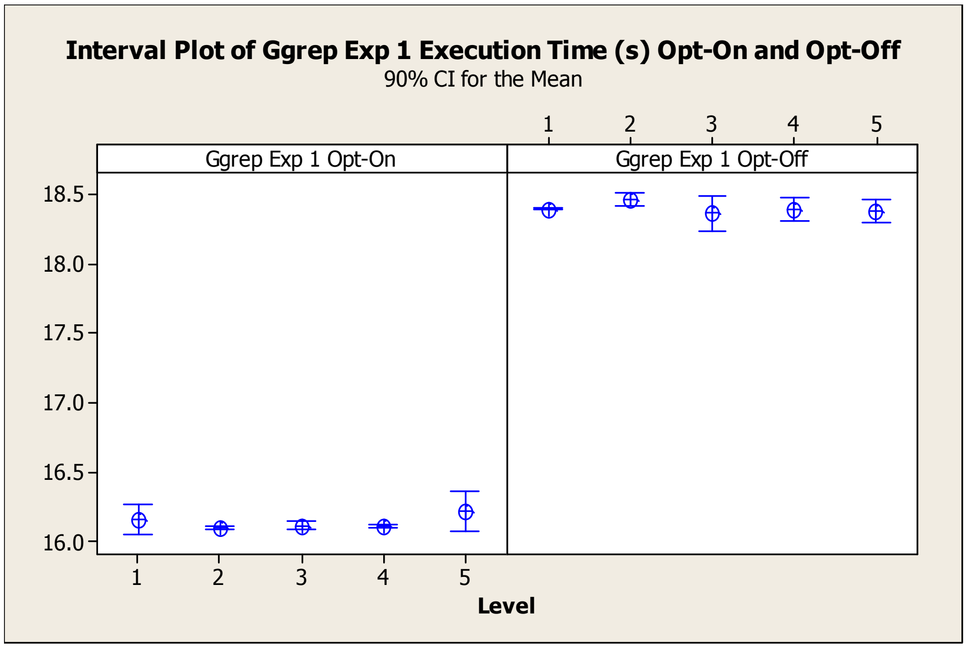


Figure A.100. Mean Interval Plot of Ggrep Expression 1 w/ IDAPro and Optimization On versus Off

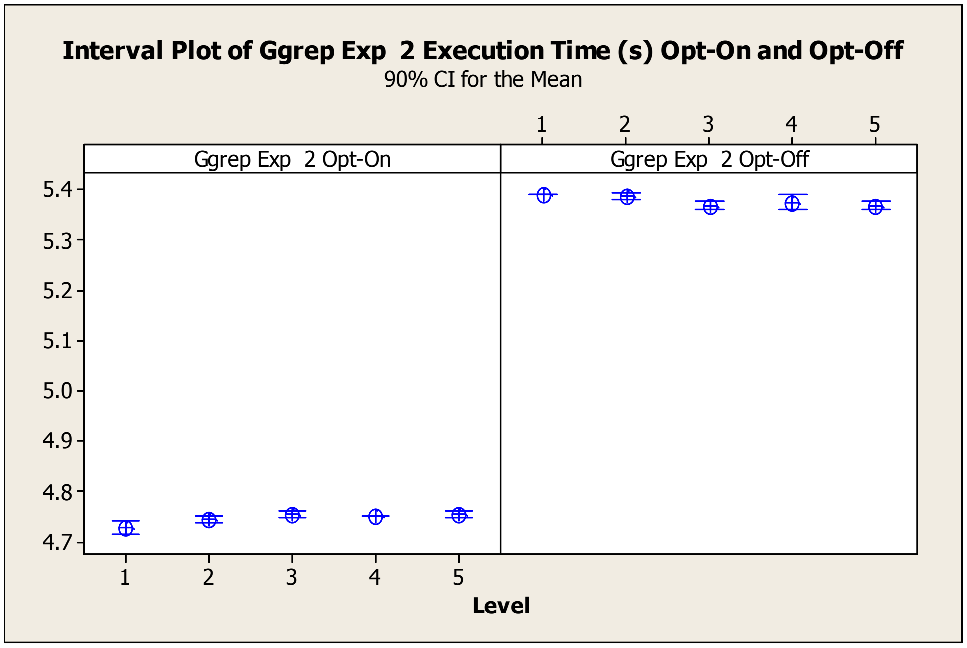


Figure A.101. Mean Interval Plot of Ggrep Expression 2 w/ IDAPro and Optimization On versus Off

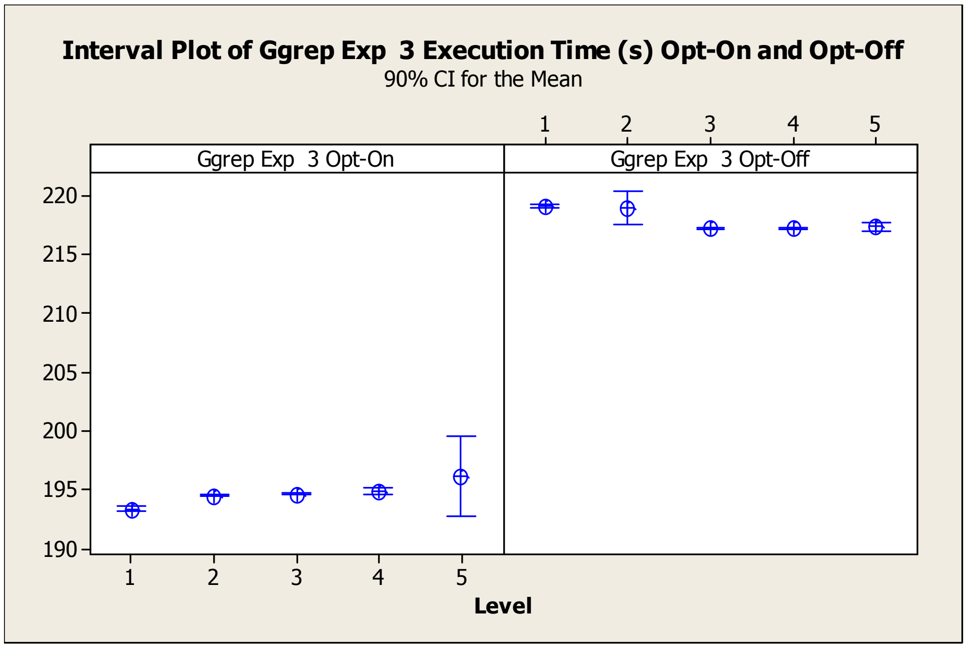


Figure A.102. Mean Interval Plot of Ggrep Expression 3 w/ IDAPro and Optimization On versus Off

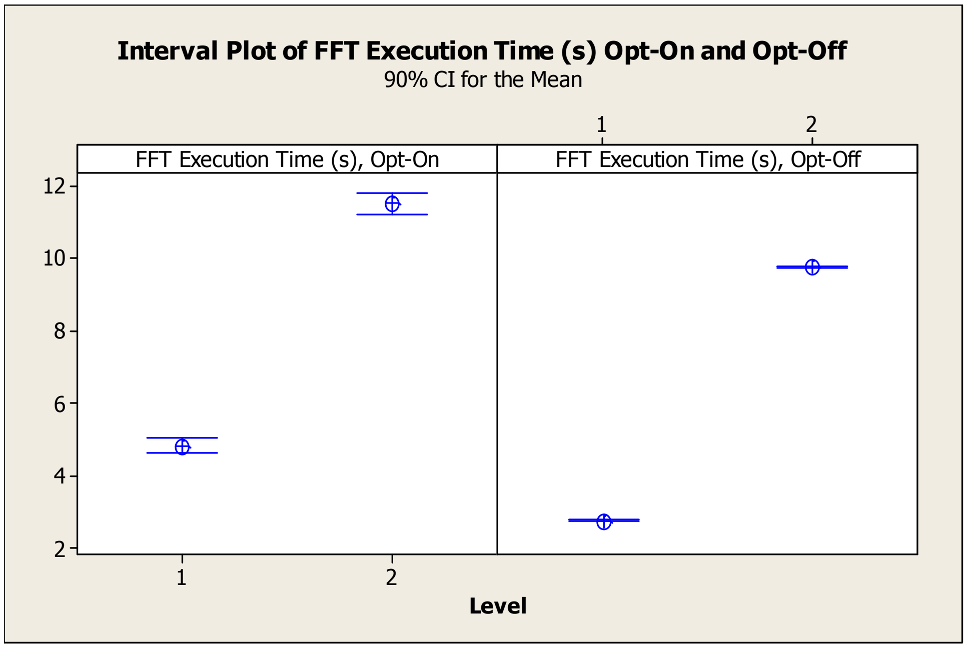


Figure A.103. Mean Interval Plot of FFT w/ IDAPro and Optimization On versus Off, Levels 1 and 2

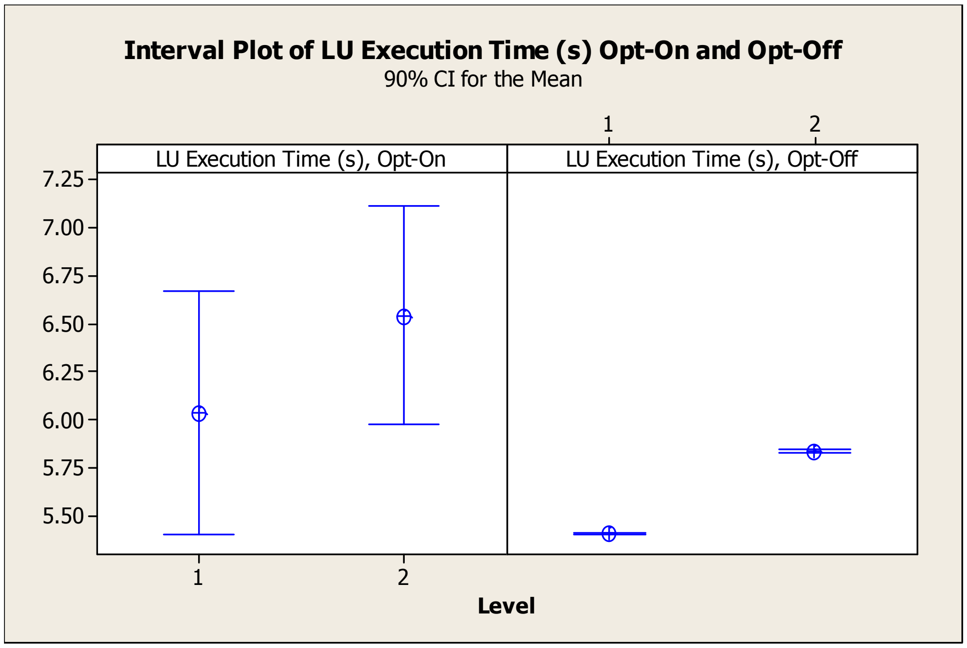


Figure A.104. Mean Interval Plot of LU w/ IDAPro and Optimization On versus Off, Levels 1 and 2

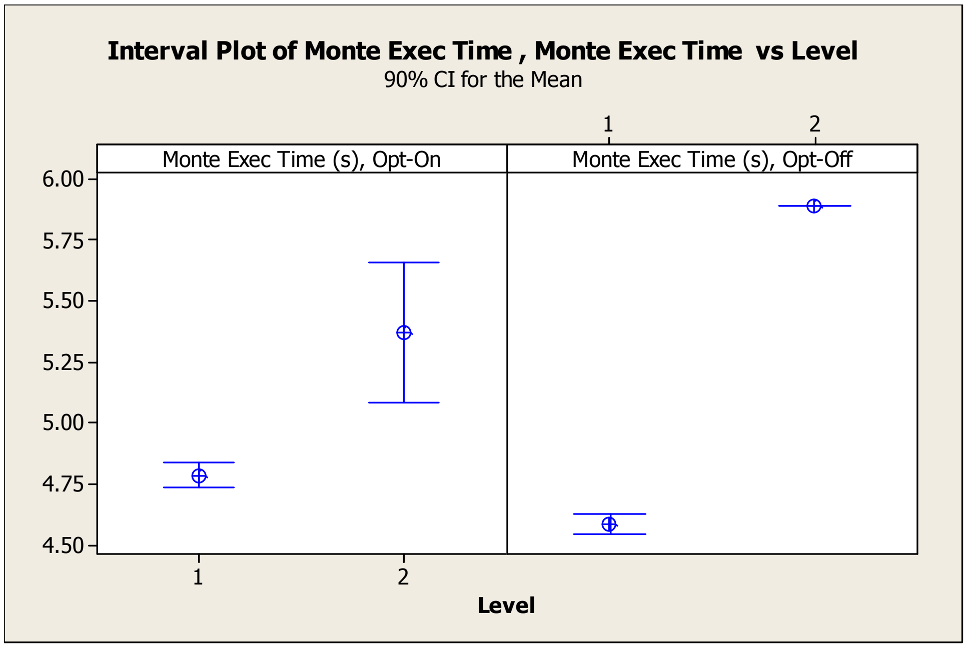


Figure A.105. Mean Interval Plot of Monte w/ IDAPro and Optimization On versus Off, Levels 1 and 2

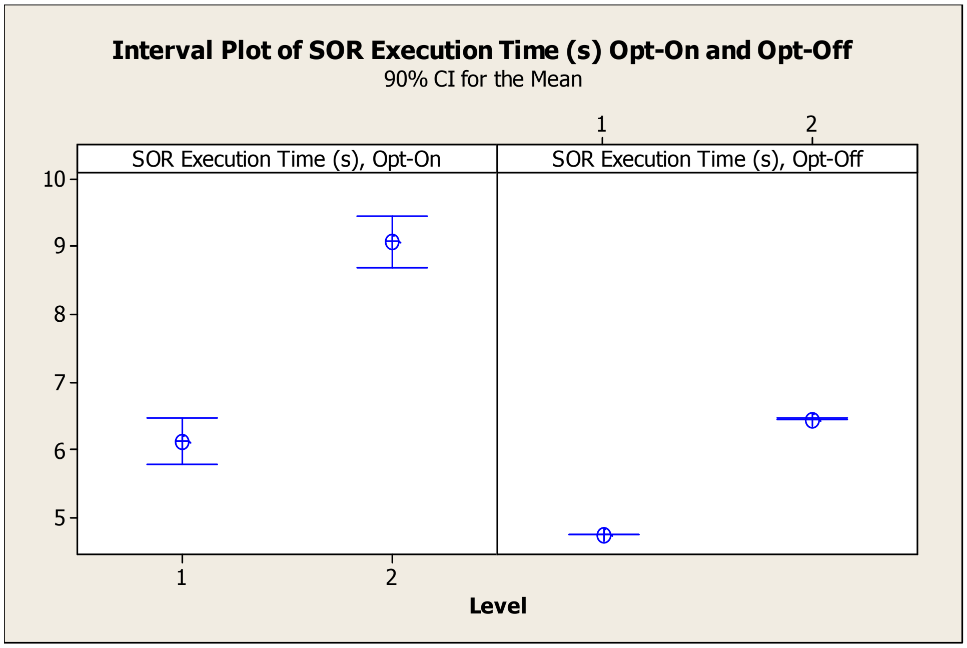


Figure A.106. Mean Interval Plot of SOR w/ IDAPro and Optimization On versus Off, Levels 1 and 2

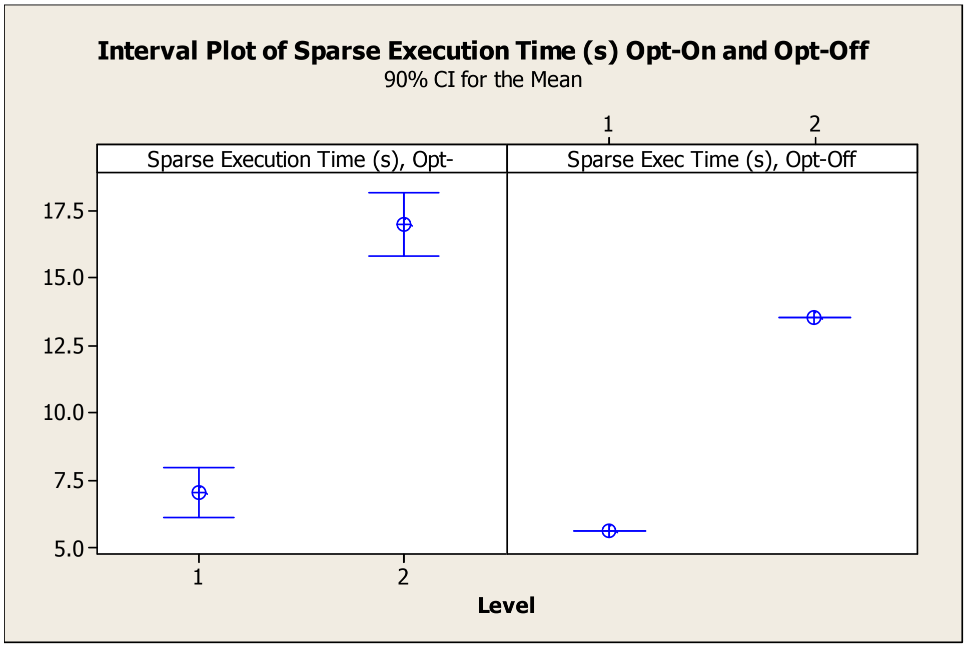


Figure A.107. Mean Interval Plot of Sparse w/ IDAPro and Optimization On versus Off, Levels 1 and 2

Table A.23. Cost per thread Analysis for SciMark2 OllyDbg and IDAPro with Optimization Off

Level	Num Threads	Function	Mean	Calls to Hidden Function	Time per Call (Mean/Call)	Time per Call/Num Threads
3	4	FFT	15456	444542000	3.47684E-05	8.6921E-06
4	8	FFT	26159	444542000	5.88448E-05	7.3556E-06
5	12	FFT	38102	444542000	8.57107E-05	7.1426E-06
3	4	LU	376.12	10700000	3.51514E-05	8.7879E-06
4	8	LU	633.3	10700000	5.91869E-05	7.3984E-06
5	12	LU	918.49	10700000	8.58402E-05	7.1533E-06
3	4	Monte	1748.8	50005000	3.49725E-05	8.7431E-06
4	8	Monte	2951.6	50005000	5.90261E-05	7.3783E-06
5	12	Monte	4324.3	50005000	8.64774E-05	7.2064E-06
3	4	SOR	12969	301974751	4.29473E-05	1.0737E-05
4	8	SOR	25460	301974751	8.43117E-05	1.0539E-05
5	12	SOR	36131	301974751	0.000119649	9.9708E-06
3	4	Sparse	25909	748500000	3.46146E-05	8.6536E-06
4	8	Sparse	44086	748500000	5.88991E-05	7.3624E-06
5	12	Sparse	64337	748500000	8.59546E-05	7.1629E-06
3	4	FFT	15131	444542000	3.40373E-05	2.8364E-06
4	8	FFT	37011	444542000	8.32565E-05	6.938E-06
5	12	FFT	39510	444542000	8.8878E-05	7.4065E-06
3	4	LU	379.87	10700000	3.55019E-05	2.9585E-06
4	8	LU	839.94	10700000	7.84991E-05	6.5416E-06
5	12	LU	1040.3	10700000	9.72243E-05	8.102E-06
3	4	Monte	1690	50005000	3.37966E-05	2.8164E-06
4	8	Monte	4312.5	50005000	8.62414E-05	7.1868E-06
5	12	Monte	4295.6	50005000	8.59034E-05	7.1586E-06
3	4	SOR	14583	301974751	4.82921E-05	4.0243E-06
4	8	SOR	25997	301974751	8.609E-05	7.1742E-06
5	12	SOR	26037	301974751	8.62224E-05	7.1852E-06
3	4	Sparse	25538	748500000	3.41189E-05	2.8432E-06
4	8	Sparse	59287	748500000	7.92077E-05	6.6006E-06
5	12	Sparse	66613	748500000	8.89953E-05	7.4163E-06
					Average =	7.04906E-06

Bibliography

[CoT98] Collberg, Christian, Clark Thomborson, and Douglas Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," *Proceedings of the Principles of Programming Languages 1998*.

[CTL97] Collberg, C., C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," *University of Auckland Technical Report*, vol. 170, 1997.

[CTL98] Collberg, C., C. Thomborson, and D. Low, "Breaking abstractions and unstructuring data structures," *Proceedings from International Conference on Computer Languages, 1998.*, pp. 28-38, 1998.

[Dub06] Dube, Thomas, "Metamorphism as a Software Protection for Non-Malicious Code," Dissertation/Thesis, 2006.

[Eib06] Joachim, Eibl. KDiff3. Ver. 0.9.91. Computer Software. http://kdiff3.sourceforge.net/ 2006. Accessed 14 Feb 2007.

[Eli05] Eliam, Eldad, *Reversing: Secrets of Reverse Engineering*. Indianapolis, Indiana: Wiley Publishing, Inc., 2005.

[GaI05] Gatlin, Kang Su and Pete Isensee. "OPENMP AND C++ Reap the Benefits of Multithreading without All the Work." http://msdn.microsoft.com/msdnmag/issues/05/10/OpenMP/ 2005. Accessed 14 Feb 2007.

[Gha04] Ghais. Ggrep. Ver. 1.0. Computer Software. www.planet-source-code.com/ 2004. Accessed 14 Feb 2007.

[Hex02] Phillips, Andrew W. HexEdit. Ver. 2.00. Computer Software. www.expertcomsoft.com/hexedit.htm 2002. Accessed 14 Feb 2007.

[Ida06] Data_Rescue. Ida Pro Disassembler and Debugger. Ver. 4.6.0.809. Computer Software. http://www.datarescue.com/idabase/ 2006. Accessed 14 Feb 2007.

[LiD03] Linn, C. and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings of the 10th Association for Computing Machinery conference on Computer and communication security*, pp. 290-299, 2003.

[Low98] Low, Douglas, "Java Control Flow Obfuscation," University of Auckland, 1998.

[MAM05] Madou, M., B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere, "Software Protection through Dynamic Code Mutation," *Proceedings of the 6th International Workshop on Information Security Applications*, pp. 371–385, 2005.

[MIN05] Minitab. MINITAB 14.0. Ver. 14.20. Computer Software. 2005.

[MVS05] Microsoft. Microsoft Visual Studio Professional 2005. Ver. 8.0.50727.42. Computer Software. http://msdn.microsoft.com/vstudio/ Accessed 14 Feb 2007.

[Oll05] Yuschuk, Oleh. OllyDbg. Ver. 1.10. Computer Software. http://www.ollydbg.de/ 2005. Accessed 14 Feb 2007.

[Ope05] OpenMP Application Program Interface. http://www.openmp.org/ Accessed 14 Feb 07.

[OSS03] Ogiso, T., Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," *Institute of Electronics, Information and Communication Engineers, Transactions on Fundamentals*, pp. 176–186, 2003.

[PoM04] Pozo, R and B Miller. "About SciMark 2.0" http://math.nist.gov/scimark2/about.html. Accessed 14 Feb 2007.

[Sci2.0] Pozo, R and B Miller. SciMark 2.0. Ver. 2.0. Computer Software. http://math.nist.gov/scimark2/download_c.html 2004. Accessed 14 Feb 2007.

[Web96] *Webster's II New Riverside Dictionary Revised Edition*. Boston, MA: Houghton Mifflin Company, 1996.

[WDH03] Wang, C., J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," *International Conference of Dependable Systems and Networks*, pp. 193-202, 2003.

[WHK00] Wang, C., J. Hill, J. Knight, and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," *University of Virginia, Charlottesville, VA*, 2000.

[Wol96] Wolfe, Michael, "High Performance Compilers for Parallel Computing," Anonymous, Ed. Redwood City, CA: Addison-Wesley Publishing Company, 1996, pp. 22-23, 137-139.

[ZhG03] Zhang, X. and R. Gupta, "Hiding program slices for software security," *International Symposium on Code Generation and Optimization,* pp. 325-336, 2003.

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 074-0188*

1. REPORT DATE *(DD-MM-YYYY)* 22-03-2007	**2. REPORT TYPE** **Master's Thesis**	**3. DATES COVERED** *(From – To)* Aug 2005 – Mar 2007

4. TITLE AND SUBTITLE Software Protection Against Reverse Engineering Tools	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Benson, Joshua, A., Captain, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Way, WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GIA/ENG/07-01
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AT-SPI Technology Office, AFRL/SNTA (POC: Robert Bennington), 2241 Avionics Circle, WPAFB, OH 45433-7320 (937) 320-9068	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Advances in technology have led to the use of simple to use automated debugging tools which can be extremely helpful in troubleshooting problems in code. However, a malicious attacker can use these same tools. Securely designing software and keeping it secure has become extremely difficult. These same easy to use debuggers can be used to bypass security built into software. While the detection of an altered executable file is possible, it is not as easy to prevent alteration in the first place. One way to prevent alteration is through code obfuscation or hiding the true function of software so as to make alteration difficult. This research executes blocks of code in parallel from within a hidden function to obscure functionality.

This method is tested on six programs; a DOS version of the UNIX grep utility and five computational functions: Fast Fourier Transfer, Successive Over-Relaxation, Sparse matrix-multiply, Monte Carlo integration, and dense LU factorization. It tests the impact of using four, eight, and twelve parallel threads of execution to obscure functionality.

The concept is effective, but is limited due to the cost associated with using threads. The computational functions make millions of calls to the hidden function. The average cost per thread for these five functions turns out to be 7.04906 x 10^{-6} seconds. The grep function does not make millions of calls and is therefore more feasible. Care must be taken to ensure the compiler does not remove parallel threads if optimization is used.

15. SUBJECT TERMS
Obfuscation, Software Protection, Parallel Threads, OpenMP, Reverse Engineering

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Rusty O. Baldwin, Civ, USAF
REPORT U	**ABSTRACT** U	**c. THIS PAGE** U	UU	170	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4445; e-mail: Rusty.Baldwin@afit.edu