USGS
science for a changing world

Efficient Processing of Two-Dimensional Arrays with C or C++

Chapter 1 of
Section E, Evaluating and Improving Computational Performance
Book 7, Automated Data Processing and Computations

Techniques and Methods 7–E1

U.S. Department of the Interior
U.S. Geological Survey

Efficient Processing of Two-Dimensional Arrays with C or C++

By David I. Donato

Chapter 1 of
Section E, Evaluating and Improving Computational Performance
Book 7, Automated Data Processing and Computations

Techniques and Methods 7–E1

U.S. Department of the Interior
U.S. Geological Survey

U.S. Department of the Interior
RYAN K. ZINKE, Secretary

U.S. Geological Survey
William H. Werkheiser, Acting Director

U.S. Geological Survey, Reston, Virginia: 2017

Suggested citation:
Donato, D.I., 2017, Efficient processing of two-dimensional arrays with C or C++: U.S. Geological Survey Techniques and Methods Report 7–e1, 58 p., https://doi.org/10.3133/tm7E1.

ISSN 2328-7055 (online)

Acknowledgments

The assistance of Dr. Qunying Huang and the Center of Intelligent Spatial Computing at George Mason University in the design of the comparative study described in this report is gratefully acknowledged.

Contents

Figures

Tables

Abbreviations

AMD	Advanced Micro Devices, Incorporated
CPU	Central processing unit
DMA	Direct memory access
GB	Gigabyte
GCC	GNU Compiler Collection
OS	Operating system
RAM	Random access memory
USGS	U.S. Geological Survey

Efficient Processing of Two-Dimensional Arrays with C or C++

By David I. Donato

Abstract

Because fast and efficient serial processing of raster-graphic images and other two-dimensional arrays is a requirement in land-change modeling and other applications, the effects of 10 factors on the runtimes for processing two-dimensional arrays with C and C++ are evaluated in a comparative factorial study. This study's factors include the choice among three C or C++ source-code techniques for array processing; the choice of Microsoft Windows 7 or a Linux operating system; the choice of 4-byte or 8-byte array elements and indexes; and the choice of 32-bit or 64-bit memory addressing. This study demonstrates how programmer choices can reduce runtimes by 75 percent or more, even after compiler optimizations. Ten points of practical advice for faster processing of two-dimensional arrays are offered to C and C++ programmers. Further study and the development of a C and C++ software test suite are recommended.

Key words: array processing, C, C++, compiler, computational speed, land-change modeling, raster-graphic image, two-dimensional array, software efficiency

Introduction

Fast and efficient[1] processing of two-dimensional arrays is essential in land-change modeling because of the substantial computational effort expended by some[2] computer land-change models in processing raster-graphic images and other two-dimensional arrays. Elapsed, wall-clock runtimes[3] for spatially explicit land-change models often span days or weeks, so improvements in the speed and efficiency of a land-change model's executable computer code can reduce its runtime by hours or even days. The speed of execution can make the difference between a model that is fast enough to be used and one that is not. Computational speed and efficiency in array-processing are fundamental, threshold concerns in land-change modeling and other areas of scientific computation.

The purpose of this report is to provide software developers with specific, practical advice on how to prepare computation-intensive software with C or C++ to achieve fast and efficient processing of large, two-dimensional arrays in random access memory (RAM). The advice covers aspects of selecting the computational environment and developing and compiling efficient C or C++ source code. This report explains the basis for the advice provided.

An explanation of C and C++ syntax for declaring arrays and accessing their elements is a prerequisite for understanding why different coding approaches result in different array-processing speeds. To meet this need, the next section of this report explains both compile-time and runtime implications of C and C++ array-processing syntax. The subsequent section describes a comparative study designed to relate computational runtimes to alternative coding approaches and other determinants of processing speed. The study is based on runtime data for a suite of test programs using 32-bit or 64-bit memory addressing under the Microsoft Windows 7 or Linux operating systems. The report presents and discusses the results and findings of the study illustrated by 40 scatter diagrams in appendix 1 and 20 boxplots in appendix 2. Based on these findings, the report presents a practical, evidence-based summary of principles for the efficient processing of two-dimensional arrays with C and C++. The report concludes with recommendations for further study and a key recommendation for developing a suite of timing-test software to assist programmers in deciding how to achieve fast and efficient processing of two-dimensional arrays.

Although this report was motivated by the need for efficiency in land-change modeling, the findings and advice apply to any C or C++ software requiring fast, serial processing of two-dimensional arrays. Moreover, this information and advice on the serial processing of two-dimensional arrays can be combined with methods of parallel processing and other computational techniques to achieve broader and more robust improvements in the speed and efficiency of computation.

---

[1]The efficiency of software is largely determined by its speed and throughput. However, software efficiency also entails conservative use of computer memory and other system resources.

[2]Although there are many land-change models that do not require intensive computation, the spatially explicit and computationally intensive land-change models–the ones that need to process two-dimensional arrays quickly and efficiently–make up an important subgroup of land-change models.

[3]The runtime is sometimes called the *makespan*.

Understanding C and C++ Syntax for Two-Dimensional Arrays

Similarities and Differences Between C and C++

Although C and C++ are different languages (Stroustrup, 2013, p. 1271–1278), they share a core syntax and many features. Since it is almost —but not quite—true that C++ is a superset of C, this report's description of the syntax for working with arrays, and with two-dimensional arrays in particular, applies to both languages. Information about classes and objects, of course, applies only to C++.

The C++ standard library provides some containers—including the `valarray`, `vector`, and `array` classes—that can be used with C++ for processing two-dimensional arrays. These containers offer advantages over standard C arrays. For example, a `vector` (though not an `array`) can grow and shrink dynamically. Another advantage of these C++ containers is the protection they provide against out-of-bounds access (Josuttis, 2012, p. 169–172, 267). Despite these advantages, the study of runtimes with these containers is beyond the scope of this report, largely because these containers are not available for C programming. A study of runtimes using these C++ containers is recommended for a follow-up study.

Standard and Alternative Array Notation in C and C++

The C and C++ programming languages provide a standard notation[4] for expressing computation with one-dimensional, two-dimensional, and higher-dimensional arrays. This standard notation allows both the declaration of arrays of elements having various built-in or user-defined data types and the creation of expressions for referencing the elements of arrays. Standard array notation in C and C++ is similar to the subscript notation of mathematics and is, therefore, natural and intuitive for many programmers. The C and C++ notation differs from mathematics in its use of square brackets rather than typographic subscripting to delineate array indexes. For example, in mathematical notation, the element in the sixth row and third column of a matrix **X** would be denoted as follows—

$$x_{6,3}$$

By contrast, in C or C++, this element would be denoted as follows—

```
x[6][3]
```

Standard notation is easy for programmers to read, write, and understand. The natural and intuitive appeal of standard notation makes array-processing code easier to maintain and may help reduce the incidence of syntax errors during development. Additionally, C and C++ compilers provide syntax checking for standard array notation that helps prevent some array-access errors during program execution, such as attempts to access array elements with too many or too few indexes.

Despite the advantages of standard notation, alternative notations for array declaration and access are available in C and C++, and there are valid reasons for programmers to choose them. Examples of alternative notations are provided later in the "Standard and Alternative Coding Techniques" section of this report. The main reasons for using alternative notation in land-change modeling are the need for faster processing and the need for programmatic control over allocation and deallocation of random access memory (RAM). Control over the allocation of RAM for arrays and the ability to release RAM used by one array and reallocate it for use by another are essential in many land-change models because of the characteristic use of many large arrays in spatially explicit land-change modeling.

Without studying actual runtime data, it may not be obvious that alternative array-processing notation might affect processing speed. However, the comparative study of runtimes establishes empirically that the choice of array-coding technique does substantially influence runtimes, although the relationship between the coding technique and runtimes is complicated by interactions with other factors. In contrast to this complex relationship, there is a clear and direct association between alternative notation and control over RAM allocation and deallocation. Standard notation allows only limited control over the allocation and release of RAM, whereas alternative notations enable greater programmatic control.

The limitations implied by standard array notation derive from the way a C or C++ compiler must generate code to accomplish the allocation of memory for any data fully specified at compile time, including standard-notation arrays.[5] With standard notation, memory for an array is allocated during program initiation, function execution, or object instantiation. Standard-notation arrays are thereby constrained as follows:

- Arrays at module scope and allocated[6] at program initiation persist until program termination.
- Arrays at module scope and allocated at program initiation are usually placed in the program's data segment, which may be limited in size by some operating systems.

[4]The term "standard notation" is not widely used in describing array syntax and notation in C or C++. The term was coined for this report to distinguish the native, built-in array syntax of C and C++ from alternative C and C++ forms and methods for declaring, allocating memory for, and accessing elements of, arrays.

[5]Most compilers store initialized and uninitialized variables in separate areas of an executable or object file. The segment for uninitialized arrays and other data is called .bss; it usually contains only one integer expressing the length the segment will have when the program is executed and the arrays and other data in the segment are then initialized.

[6]The usage "an array is allocated" is an informal substitute for "space in memory is allocated for an array."

- Arrays allocated within function scope can either be static, and therefore allocated in the potentially size-limited data segment, or automatic,[7] and therefore allocated on the call stack.[8] The call stack is typically limited in size by operating systems.
- Arrays allocated at object instantiation can be allocated in the data segment or on a stack.

These constraints lead to specific software problems:

1. Programs may terminate abnormally during array allocation because of limitations on stack size, data-segment size, or available RAM.
2. Large arrays allocated statically cannot be removed from RAM to free up needed memory.

The first problem can be mitigated through operating-system specific measures. For example, in Linux, the data-segment size and stack size can be increased during program execution with the `setrlimit()` function, although not beyond the hard limits built into the Linux kernel. The second problem is a necessary feature of standard array notation that can only be avoided by using an alternative notation that allocates memory dynamically.

How C and C++ Compilers Process Arrays

When a C or C++ compiler processes a declaration including a pair of square brackets, as with

```
int myVector[200];
```

conceptually, the compiler performs three actions:

1. allocates memory to store 200 `int` values;
2. construes the type of `myVector` as `int *` in expressions throughout the compilation unit; and
3. assigns the memory address of the first element of the array `myVector[200]` to `myVector`.

Depending on the compiler and the scope of a declaration (module, function, or class), the elements of an integer array, for example, may also all be initialized to zero (0) immediately after allocation.

Although the type of `myVector` is treated by the compiler as pointer to integer (`int *`), the name and token `myVector` cannot be used in source code as if explicitly declared as a pointer, unlike (for example) the name `intptr` in this example of an explicit pointer declaration:

```
int *intptr;
```

The compiler does not allow `myVector` to be used as an l-value, and it only allows `myVector` to be used in an r-value expression if it is followed immediately by a pair of square brackets enclosing a valid index expression.

When a compiler processes an expression containing `myVector[indexvalue]` (where `indexvalue` is an expression that evaluates to an integer of the required size), the compiler creates code to perform the following actions:

1. evaluate the index value;
2. multiply the index value by the size of the type to which `myVector` points (in this case the size would be `sizeof(int)`);
3. add the product from the previous step to the address contained in `myVector` (with `myVector` of type `int *`); and
4. dereference the address that is the sum from step 3, making the `int` stored at this address available in the context of the expression containing `myVector[indexvalue]`.

C and C++ syntax require only that the token or expression to the left of square brackets (`[…]`) simplify to a pointer to some specific data type. When the C or C++ compiler encounters square brackets, it creates code to compute an address in memory by using the index value from the expression or constant within the square brackets, along with the value and type of the pointer identified by the token or the expression to the left of the brackets. The computed memory address is used to make a value from RAM available for use within the context of the expression. This feature of C and C++ syntax allows programmers to instruct the compiler to generate code that performs the same kind of address arithmetic used with standard-notation arrays. Programmers may thus make use of square-bracket notation to access any value within any contiguous extent of allocated RAM, whether the extent is allocated through an explicit, standard-notation array declaration or not. For example, square-bracket notation may be used for array-element access within a contiguous block of RAM that was allocated dynamically using the C `malloc()` function or the C++ `new` operator.

In an expression for a multidimensional array, such as a two-dimensional array `matrix[a][b]`, the token `matrix[a]` must resolve to a pointer type since the compiler requires the expression or token to the left of `[b]` to be of pointer type. Moreover, `matrix` must resolve to a pointer type since the compiler expects the expression or token to the left of `[a]` to be a pointer. Thus, if the declaration for this array is

```
int matrix[200][100];
```

then

- `matrix` is of type `int **;`
- `matrix[a]` is of type `int *;` and
- `matrix[a][b]` is of type `int`.

[7] Data declared within a function without the storage-class specifier `static` are automatically allocated during execution of the function and automatically de-allocated on exit from the scope of the function. The storage class of such data is the "automatic" class.

[8] Some C and C++ compilers differ in their methods for allocating RAM. The C and C++ language specifications do not require that automatic variables be allocated on a stack.

If `matrix` had been declared as
```
int *matrix[200]; //matrix is an array of
//200 pointers to type int
```
and then initialized so that each element of `matrix` points to a 100-element array of integers, then the same syntax (`matrix[a][b]`) may be used to refer to individual integers (Myers, 2014, p. 15–18).

These observations about how compilers handle the syntax of square-bracket notation reveal a key fact applied in designing code for computational efficiency: a reference to an element of a two-dimensional array implies four arithmetic operations to compute the memory address of the array element. The four operations are as follows:

a. two integer multiplications of an index value by a type size in bytes (one multiplication for each index); and

b. two integer additions of a pointer and the pointer offset value produced by one of the multiplications in Item a (one addition for each index).

McConnell (2004, p. 602), provides additional perspective on the computational costs associated with array-element access.

Design of a Comparative Factorial Study of Runtimes

Standard and Alternative Coding Techniques

Three basic coding techniques for array processing were included in the comparative study. These techniques are—

1. **Standard Notation for Arrays**
   - Array declaration and allocation: `int array[8000][8000];`
   - Array element access: `int i,j;… array[i][j] …`
2. **Programmer Specification of Array Addressing**
   - Array declaration and allocation: `int array[64000000];`
   - Array element access: `int i,j;… array[i*8000+j] …`
3. **Programmer Array Allocation with Standard Access Notation**
   - Array declaration: `int **array;`
   - Array allocation: `int i;array = malloc(8000 * sizeof(int *));for (i=0; i<8000; i++){array[i] = malloc(8000 * sizeof(int));}`
   - Array element access: `int i,j;… array[i][j] …`

In these coding paradigms, `i` serves as the row index and `j` as the column index. The integer values 8000 and 64000000 are used as array dimensions for consistency with the array dimensions in the suite of test programs described in the following subsection.

Especially in C++, some experts discourage the use of standard notation for multidimensional arrays because of the risk of errors during program execution such as out-of-bounds index values (Stroustrup, 2013, p. 974–977). The C++ standard library provides containers safe from such errors, but these containers are unavailable for C programming. The advice is worthy of notice but should not prevent developers from developing specialized code to meet the performance requirements of their applications.

The Suite of Test Programs

A suite of 28 small C test programs and 20 small C++ test programs was developed to compare the relative computational speed of three basic C and C++ source-code techniques for two-dimensional arrays[9] and to measure the effects of other factors on computational speed. Each program performs the same amount of computational work as the other programs, but the way the source-code expresses the computation varies from program to program. For comparison and analysis, the time in seconds required to run each of the 48 programs was measured and recorded for multiple runs of the programs. The times measured were wall-clock elapsed times (meaning throughput times, or makespans), not central-processing-unit (CPU) usage times.

The test programs perform limited computation within loops that access every one of the 64,000,000 elements of an array exactly once. Computation within the loops is limited so that most computational work in each test program consists of determining array-element addresses and other computational overhead for loop processing. The arrays consist of integers because integer arithmetic is faster than floating-point arithmetic. The use of floating-point computations could confound the tests by introducing lengthy computation times that could mask differences due to loop processing overhead and array-element access.

Appendix 3 contains the source code for the 28 C test programs. Appendix 4 contains the source code for the 20 C++ test programs. Comparing these short source-code files is encouraged.

Each of the 48 programs in the test suite declares and allocates a two-dimensional array of 8000×8000 integers.

[9]The comparative study was designed with the expectation that coding techniques would be one of the strongest runtime determinants. That expectation was not borne out by the study. Although the study shows that the choice of coding technique strongly influences runtimes, coding-technique influence is confounded by interactions with other factors and sometimes overshadowed by stronger factors.

In some programs, the type of array element is `int` and in others it is `long int`. For the hosts and compilers used in the comparative study, the size of an `int` is 4 bytes, and the size of a `long int` is 8 bytes. Therefore, each array has 64,000,000 elements and requires either 256,000,000 bytes of RAM or 512,000,000 bytes—these array sizes are approximately 244.14 mebibytes and 488.28 mebibytes, respectively. This array size was chosen because it is within the range of array sizes encountered in land-change modeling and because it is suitable for timing tests that require between 5 and 180 seconds of wall-clock time, which is just long enough for valid comparisons.

All three coding techniques set out in the preceding subsection were tested in C programs, but only techniques 2 and 3 were tested in C++ programs because each of the C++ test-suite programs encapsulates a large two-dimensional array within an object. An object cannot be successfully instantiated when it attempts to allocate a sufficiently large array declared with the standard notation used in coding technique 1. Although code for technique 1 (with an array declared at module level rather than encapsulated in an object) can be compiled with a C++ compiler, the results would be the same as for a comparable C program and would not provide additional information for comparing coding techniques.

The C++ implementations of techniques 2 and 3 differ from the C implementations, but they are operationally equivalent. In C++, array elements encapsulated within an object are accessed through inline object member functions and the `new` operator is used rather than `malloc()` to allocate RAM for arrays. The C++ implementations do not use containers from the C++ standard template library, such as `array` or `valarray`[10] or `vector`, although a future study to test performance with these containers is recommended.

For each of the three basic coding techniques, there are multiple programs in the test suite, and each provides a unique variant of the basic coding technique. The variants involve (i) the size of the array elements (`int` or `long int`), (ii) the size of the array indexes (also `int` or `long int`), and (iii) specification or nonspecification of the `register` storage class for array indexes. There are 4 variants of basic coding technique 1, 5 variants of basic coding technique 2, and 5 variants of basic coding technique 3, meaning there are 14 variants among the C test programs (4 + 5 + 5) and 10 variants among the C++ test programs (5 + 5).

Each variant has two forms: a straightforward, sequential, unrandomized loop over an array, proceeding row-by-row and then column-by-column within each row, and a randomized loop that processes an array in a sequence that cannot be deterministically predicted at compile time. The total number of C test programs is 28 (14 unrandomized and 14 randomized), and the total number of C++ test programs is 20 (10 unrandomized and 10 randomized).

[10]The valarray container does not clearly provide a suitable replacement for standard C arrays because of the lack of support for this container among compiler developers (Josuttis, 2012, p. 943).

Differences Among Compilers

A limitation of the comparative study is the use of only one C/C++ compiler under Linux (GCC) and one under Windows (Visual C/C++). The study could be improved by replicating the runtime tests under at least one additional Linux C/C++ compiler and at least one additional Windows C/C++ compiler. Possible choices include the C language family of LLVM compilers (University of Illinois LLVM Project, 2016) for Linux and the Intel C++ Compilers (Intel, 2016) for Windows and Linux.

Other Factors Affecting Computational Speed

The effects of four additional factors on runtimes for the 48 test-suite programs were also measured:

1. Compiler optimization level (no optimization or moderate optimization)
2. Host hardware (one of three systems used for testing, as shown in table 1)
3. Memory addressing model (32-bit or 64-bit)
4. Operating system (Linux or Windows 7)

Whether the 32-bit memory addressing model is becoming obsolete is far from clear. At the time of publication, a substantial proportion of working computer systems use 32-bit operating systems, and a few of these operating systems run on processors lacking hardware support for 64-bit addressing. Although the mainstream processors used in new desktop and laptop computers have been 64-bit capable for over a decade, both the Windows and Linux operating systems remain available in 32-bit and 64-bit versions. Because of the continuing availability and use of 32-bit memory addressing, and because many land-change and other models can run on systems using 32-bit memory addressing, the choice

Table 1. Descriptions of host hardware used in the comparative study.

Host identifier	Host description (GB, gigabyte; RAM, random access memory)
A	Personal workstation with dual Intel Xeon quad-core E5440 processors and 4.0 GB of RAM.
B	Personal workstation with AMD Phenom II dual-core Model 555 processor and 8.0 GB of RAM.
C	Personal workstation with AMD Phenom II quad-core Model 965 processor and 16.0 GB of RAM.

between 32-bit or 64-bit memory addressing is included in the comparative study as a factor. Arguably less important, but worth consideration, is the potential for use of low-power, inexpensive 32-bit processors, such as those developed for mobile or special-purpose devices, repurposed and linked into massively parallel computational clusters.

Hardware Considerations for Software Performance

The comparative study accounts for the effect of computer hardware only at the level of complete systems, not at the level of components such as processors, hard disk drives, and RAM modules. These determinations are accomplished simply by running the test programs on three different hosts. The resulting data enable comparison of the relative—but not absolute—effects of the other processing-speed determinants among the three hosts. For practical purposes, in many of the situations faced by programmers, the finding that the *relative* importance of the determinants of processing speed varies from host to host is sufficient to inform programmers of the need to adjust and customize code for each host.

This is not to suggest that programmers never need to account for the specific details of host hardware when writing code; programmers regularly improve software performance for individual hardware configurations. For example, a programmer might determine how many memory-page faults occur—while the code is executing—to fine-tune a processing loop. In this example, the programmer might alter the code to perform periodic block transfers of part of an array into a small, temporary array before continuing with loop processing. This code alteration could speed-up overall processing by tuning the software for effective use of memory caches located on the processor chip, thereby replacing many instances of RAM access with instances of order-of-magnitude faster access to cache memory. The code alteration might also tune the software to make better use of the direct memory access (DMA) capabilities of the host. Although most compilers provide similar optimizations, including some specific to Intel or AMD processors, programmers still enjoy the advantage over compilers of being able to study software performance on specific hardware during program execution. This advantage allows programmers to improve on compiler optimizations.

The list of hardware components with direct effects on processing speed includes at least the following—

- processor make, model, stepping, clock rate, and data-path size;
- available processor cores, registers, and extensions;
- availability of pipelining features such as hyper-threading;
- size and availability of processor caches;
- make, model, and version of the chipset supporting RAM access;
- operational characteristics of the front-side bus, HyperTransport, or other path between processor and RAM;
- relationship between the processor-to-RAM bus or pathway, and peripheral buses or data pathways; and
- type and features of RAM hardware (for example, whether RAM is buffered or not, or protected by error-correcting codes or not).

The study only compares hardware at the aggregate (or host) level, because otherwise the number of comparisons and contrasts among hardware factors would be overwhelming. The comparative study is like a map that shows how to get to a baseball stadium on the other side of town, but not necessarily how to find a specific seat within the stadium.

The Comparative Study as Factorial Experiment

The comparative study measured runtimes for various combinations of discrete levels of factors. The resulting comparative study is a factorial experiment that shows many contrasts among runtimes at discrete levels of multiple factors. Statistical methods used for the analysis of factorial experiments are not applied in the comparative study because runtimes are regarded as deterministic rather than stochastic measures, despite the observed variability among replicates described later.

Table 2 lists the factors and the discrete factor levels included in the data collected and analyzed in the comparative study. As explained in the next section, the data include redundant factors. Consequently, the comparative study analyzed the effects of a subset of the factors described in table 2. The data used in the study are provided in the USGS published data release, “Runtimes for Tests of Array-Processing Speed” (Donato, 2017).

Table 2. Fields of the runtime data files.

[All 18 rows of this table are useful in understanding the data analyzed in the comparative study, so they are retained here to document the data in Donato (2017). The ten rows corresponding to discrete determinants of runtimes are shaded in blue to distinguish them from rows of merely documentary value. Row 18, the processing time (or runtime), is the response (or dependent) variable]

Field number	Column name	Field description	Level (discrete value)
1	SystemID	Host Identifier: An alphabetic code identifying the specific hardware platform on which a processing speed was measured.	A—Intel Xeon B—AMD Phenom II Model 555 C—AMD Phenom II Model 965
2	ProcMake	Processor Make: A code indicating the make (manufacturer) of the processor	A—AMD I—Intel
3	ProcCores	Processor Cores Count: The number of processor cores on the hardware host	2 4 8
4	ProcArch	Processor Architecture: An indication of whether the processor is 64-bit-capable or not	32—32-bit processor 64—64-bit-capable processor
5	OS	Operating System: A code indicating the operating system under which speed tests were run	Windows Linux
6	OSaddmod	Operating System Address Model: A code indicating whether the OS uses the 32-bit or 64-bit address model.	32 64
7	Virtual	Virtualization Indicator: A code indicating whether the host was virtual or not	No Yes
8	Compiler	Compiler: The name and version of the compiler used	Visual Studio 2010 GCC–4.2 GCC–4.5.1
9	TestID	Test Identifier: The standardized name of the source code for the test	at1a.c, at1aR.c, at1b.c, at1bR.c, …, at3e.c, at3eR.c
10	RandomYN	Randomization Indicator: An indication of whether the array processing is purely sequential or in random sequence.	N—Unrandomized Y—Randomized
11	Language	Language Indicator: An indicator of whether the test was coded and compiled in C or in C++	C C++
12	CodeTech	Coding Technique: The basic coding technique used in the test	1—Standard array syntax (C only) 2—User index computation 3—Dynamic array allocation
13	RegUsage	Register Usage: An indicator of whether array indexes are explicitly declared in storage class “register” or not.	Y—Declared “register” N—Not declared “register”
14	ArraySz	Array Element Size: The size in bytes of each element in the raster (array) being processed	4 8
15	IndexSz	Index Size: The size in bytes of each variable used as the index of the raster (array) being processed.	4 8
16	CodeModl	Code Address Model: An indicator of whether the executable code is 32-bit code or 64-bit code. (32-bit code can be executed on a 64-bit processor and OS, but 64-bit code can only be executed under a 64-bit OS running on 64-bit hardware.)	32–32-bit OS 64-64-bit OS
17	OptLevel	Optimization Level: An indicator of whether the code was optimized for speed or not	N—No optimization (O0) O—Optimization for speed (O2)
18	ProcTime	Processing Time: The observed processing time for the test in seconds	

Analysis of the Results of the Comparative Study

Although the set of runtime data records for the comparative study includes 18 factor fields, as shown in table 2, not all the factors are used in the study, and not all factors are independent. Some factors are not used because the final dataset is more limited in scope than originally envisioned. Other factors are unused because of the redundancy originally built into the data records to simplify their analysis.

The following information may only be of interest for studying or analyzing the data in the associated data release (Donato, 2017).

- The Test Identifier, Randomization Indicator, and Coding Technique are mutually dependent because the naming convention for the Test Identifier makes the Randomization Indicator and Coding Technique redundant;
- The Host Identifier uniquely determines the Processor Make, Processor Cores Count, and Processor Architecture;
- Values for the Operating System Address Model field are the same as the values for the Code Address Model field in the data records used in this study (although these fields would not necessarily have to be identical);
- The differences between Versions 4.2 and 4.5.1 of the GCC compiler were assumed negligible and the Compiler field was treated as implied by the Operating System field, since the Visual Studio 2010 compiler was the only compiler used under Windows in this study, and GCC was the only compiler used under Linux in this study; and
- The Virtualization Indicator was not used because the data do not include records for virtual systems, contrary to original plans.

Out of the 18 factor fields included in the dataset for the study, only 10 factor fields are used:

Field number 1: Host Identifier
Field number 5: Operating System
Field number 10: Randomization Indicator
Field number 11: Language Indicator
Field number 12: Coding Technique
Field number 13: Register Usage
Field number 14: Array Element Size
Field number 15: Index Size
Field number 16: Code Address Model
Field number 17: Optimization Level

The analysis of the data shows that the speed of processing for two-dimensional arrays, with code created by C or C++ compilers, is highly variable in response to changes in the levels of these 10 factors. Data from the comparative study further show that the suite of test programs, each performing the same amount of computational work in processing two-dimensional arrays, produced runtimes ranging from a low of about 7 seconds to a high of about 160 seconds; see table 3. In order to understand how the 10 factors affected runtimes, without getting tangled in the combined effects of multiple factors, the effects of each factor (or in a few cases, a small group of related factors) are considered individually in this section. This section ends with a summary of the small set of factor levels associated with the shortest runtimes for each of the three host systems in the study.

Appendixes 1 and 2 provide scatter diagrams and boxplots to illustrate the effects of some factors, and the data analyzed in the comparative study are provided in the associated data release (Donato, 2017). For a deeper understanding of the effects of the various factors on runtimes, and to verify the summaries of factor effects provided in this section, readers should consult the appendixes and explanations contained therein for the scatter diagrams and boxplots.

How Comparisons Are Analyzed

Many interpretations of the data reported in this section were determined by sorting and re-sorting the data file in Donato (2017) on a variety of combinations of the columns. Changing the sequence of the records enables viewing specific contrasts among the 10 factors. For example, after sorting by System ID, then Language, and then Processing Time, the range of Processing Times (runtimes) can be compared among the groups of records for each combination of System ID and Language. When viewing the data with spreadsheet software, the average Processing Time for each Language and System ID can be readily computed and compared.

Variations in Runtimes

Before considering the effects of each factor or group of factors in turn, the precision of the observed runtimes, in seconds, should be understood. While the runtime for any specific combination of factor levels is conceptually fixed and repeatable (meaning deterministic rather than stochastic), the observed runtimes are subject to variation from run to run as a result of hard-to-control conditions within the computer operating environment, such as resources used by other processes on the system. To obtain runtime observations that are reliable measures of the underlying, ideal runtime, three runs were made and timed for each combination of factor levels. Although the minimum runtime for a set of three replicates is the time closest to the ideal runtime achievable on a system—when the test program does not have to contend with other processes on the system—the average of the three replicates of runtime for a particular combination of factor levels is used because it provides a more realistic measure of throughput time. Table 4 summarizes the 14 data files collected for the comparative study, showing for each file the average and maximum percent deviation of runtimes over all sets of three replicates.

Table 3. Shortest and longest runtimes by host, address model, operating system, and language.

[This table illustrates the wide ranges of runtimes observed for each of the three hosts (A, B, and C). Although comparisons between hosts are not valid, this table provides access to valid contrasts among the address model, the operating system (and by implication, the compiler), and the programming language. None of these factors stands out because of consistently high or consistently low runtimes]

Host identifier	Address model	Operating system	Language	Shortest runtime (seconds)	Longest runtime (seconds)
A	32	Linux	C	11.35	69.57
A	32	Linux	C++	11.64	115.53
A	32	Windows	C	11.63	84.73
A	32	Windows	C++	11.65	130.93
A	64	Linux	C	11.36	101.81
A	64	Linux	C++	11.65	155.65
A	64	Windows	C	11.64	84.79
A	64	Windows	C++	11.66	130.95
B	32	Linux	C	11.51	63.68
B	32	Linux	C++	8.53	135.06
B	64	Linux	C	10.12	80.72
B	64	Linux	C++	9.32	160.24
C	32	Linux	C	10.59	60.48
C	32	Linux	C++	7.29	127.99
C	32	Windows	C	7.12	68.48
C	32	Windows	C++	7.67	136.68
C	64	Linux	C	9.65	77.41
C	64	Linux	C++	7.82	152.57
C	64	Windows	C	7.13	68.03
C	64	Windows	C++	7.75	136.04

Table 4. Average and maximum percent deviations of runtimes.

[The runtime data for the comparative study were originally collected into 14 datasets. Although, to facilitate analysis, each of the datasets for runs made under the Windows 7 operating system was later divided into two datasets (one for each operating system address model), the average and maximum percent deviations are presented for the original 14 datasets. This presentation preserves effects possibly caused by the time and circumstances of the original measurement of runtimes. Times were measured in seconds. The percent deviation is computed for each set of three runtimes (replicates) measured for each unique combination of factor levels. The percent deviation for a set of three runtimes is computed as follows: (i) sum the absolute values of the three differences between each runtime and the average of the three runtimes; (ii) divide this sum of differences by 3.0 to obtain an average absolute deviation; (iii) divide the resulting quotient by the average of the three runtimes to yield the proportionate average deviation; and (iv) multiply the proportionate average deviation by 100.0 to obtain the percent deviation. In this table, the column for Average Percent Deviation contains the percent deviation averaged over a dataset, and the Maximum Percent Deviation is the highest percent deviation observed in a dataset for any single set of three runtime replicates]

Host identifier	Operating system address model	Operating system	Language	Average deviation (percent)	Maximum deviation (percent)
A	32-bit	Linux	C	0.088	0.946
A	32-bit	Linux	C++	0.066	0.266
A	32-bit and 64-bit	Windows 7	C	0.281	1.366
A	32-bit and 64-bit	Windows 7	C++	1.184	6.162
A	64-bit	Linux	C	0.072	0.418
A	64-bit	Linux	C++	0.167	3.179
B	32-bit and 64-bit	Linux	C	0.326	3.820
B	32-bit and 64-bit	Linux	C++	0.184	4.029
C	32-bit	Linux	C	0.155	2.256
C	32-bit	Linux	C++	0.090	0.513
C	32-bit and 64-bit	Windows 7	C	1.427	6.774
C	32-bit and 64-bit	Windows 7	C++	1.006	5.644
C	64-bit	Linux	C	0.897	15.473
C	64-bit	Linux	C++	0.077	0.364

The data in table 4 show that the values in most of the groups of three replicated timing runs are firmly within 1 percent of each other, but there are some cases in which the values differ by a few percentage points. The most volatile three-measurement runtime set has values that differ from one another by up to 15 percent. Eleven of the fourteen measured sets have average deviations under 1 percent, and the remaining three have values above 1 percent. The highest average deviation is 1.427 percent. The measurements for Windows are consistently more variable than comparable measurements for Linux.

Factors with the Greatest Effects on Runtimes

The runtime data identify the two factors with the greatest effects on runtimes as (i) compiler optimization and (ii) randomization of the order of access to array elements. These effects are illustrated by the 40 scatter diagrams in appendix 1 and the 20 boxplots in appendix 2.

As the data and the diagrams indicate, compiler optimization results in at least a doubling of computational speed (defined as at least a halving of runtimes). In many cases compiler optimization increases computational speed by factors up to, and even exceeding, five (for example, see figure 1–3 in appendix 1.) Compiler optimization is the most effective way to increase computational speed and reduce runtimes, although the effects of compiler optimization on runtimes vary considerably because of other factors, notably randomization of order-of-access to array elements.

As the scatter diagrams and boxplots also show, randomizing the order-of-access to array elements in a computational loop increases runtimes by a factor varying from approximately 1.5 to more than 2. Randomization precludes some compiler optimizations and reduces the speed-up due to compiler optimizations. The fact that randomization does not preclude all compiler optimizations is apparent, because when the runs are randomized, the optimized variants consistently have shorter runtimes than the unoptimized variants. The fact that randomization does preclude some compiler optimizations is also apparent, because when runs are optimized, unrandomized runtimes are consistently shorter than randomized runtimes.

It may be tempting to conclude, based on the large magnitude of speed increase due to compiler optimization, that there is no need for programmers to seek additional optimizations. *This conclusion would be incorrect* because coding techniques and other factors substantially affect the runtimes of compiler-optimized C and C++ executable code for processing two-dimensional arrays. For example, on Host A, the ratio of the longest optimized runtime to the shortest optimized runtime exceeded 3.94 as a result of other factors, including coding techniques. Compiler optimizations are necessary but not sufficient to achieve the shortest runtimes.

Effects of Coding Techniques on Runtimes

Changing the coding technique usually results in a change in the runtime, but the magnitude of the change varies with other factors. No single technique produces the shortest runtimes in all circumstances (meaning for all combinations of factor levels), but techniques 2 and 3 lead to shorter runtimes than technique 1 (standard notation) for the majority of factor combinations. There are some combinations of factors for which technique 1 produces the shortest runtimes. Please refer to “Standard and Alternative Coding Techniques” for the description of coding techniques 1, 2, and 3.

Making the correct choice of coding technique needed to achieve the shortest runtime for a specific processing loop within a program requires the programmer to test alternative versions of the loop on the target host (the system the program is intended to run on) under the conditions expected during production runs. Alternatively, programmers may consult the data in Donato (2017) for conditions similar to those expected on the target host for production runs and select the technique with the shortest runtime for those conditions.

Effects of Language Choice on Runtimes

The relative speed of compiled C code compared with compiled C++ code depends on various factors, most notably the choice of processor. On test Host A with an Intel processor, the C compilation produced the fastest code for unrandomized loops, but the C++ compilation produced faster code for randomized loops. These findings are the opposite of the findings for test Host B with an AMD processor, where the C++ compiled code was fastest for unrandomized loops, but the C compiled code was faster for randomized loops. On test Host C, with a different AMD processor than Host B, the C compilation produced faster code for both randomized and unrandomized loops than the C++ compilation.

The scatter diagrams in appendix 1 show that compiled C and C++ codes operate with nearly equal speed for many sets of factors, but for several sets of factors, C++ code runs slower than equivalent C code. Despite the slow processing of C++ code under some circumstances, runtimes for C++ code include some of the shortest for all three systems. On test Host A and test Host C, the fastest C++ runs require less than 3 percent more time than the fastest C code. On test Host B—where C++ runs are the fastest—the fastest runtime for C code is more than 18 percent longer than the best C++ runtime.

Effects of Processor Architecture on Runtimes

The preceding discussion of coding techniques and language choice indicates that the combination of factors leading to the shortest runtimes is not the same on any of the three test hosts in the comparative study. The study data show that the choice of host, and therefore the host’s processor architecture, affects how a program must be coded to achieve

the shortest runtimes. The processors for Host B and Host C belong to the same processor family, the AMD Phenom II family, yet the choices of factors producing the shortest runtimes on one of these hosts are not the same as the choices of factors required to produce the shortest runtimes on the other. Because of hardware differences among the hosts, *the runtimes for test programs on one host cannot be compared with runtimes for another host*.

Effects of Operating System and Compiler on Runtimes

Test runs were made with Linux and Windows 7 on Host A and Host C but not Host B. The ranges of runtimes for Linux and Windows 7 are not strikingly different from one another on either Host A or Host C. On Host A, the shortest runtime for code generated by Visual Studio 2010 (the compiler used with Windows 7 in the comparative study) was less than a half-second longer than the best time for Linux with GCC (the compiler used with Linux in the comparative study). On Host C, the shortest runtime for Windows was 0.169 second faster than the shortest runtime for Linux with GCC on this host, and the longest runtime for Windows on Host C was more than 15 seconds less than the longest time for Linux on this Host. On Host C, the fastest runtimes were achieved by C++ code under Linux and by C code under Windows.
These comparisons show—

1. The choice of an operating system and a compiler strongly affects runtimes.

2. There is no clear pattern across hosts to suggest that one operating system or compiler is consistently better than another.

Clearly, other factors have a stronger influence on runtimes than the operating system and compiler choice, and these other factors overshadow the effects of the operating system and compiler.

Effects of Memory Model on Runtimes

The average runtime for test programs compiled for execution under a 32-bit operating system was 40.02 seconds. The average runtime for test programs compiled for execution under a 64-bit operating system was 43.22 seconds. These values indicate that in this comparative study, faster runtimes are usually achieved with the 32-bit memory model. Although this result cannot be generalized to other hosts, or processing involving floating-point computations, nor to a different mix of integer computations, these results show that computation under the 32-bit memory model can be faster than the same computations under the 64-bit memory model. The resulting conclusion is that reverting to the 32-bit model is an option to be considered when the fastest processing of two-dimensional arrays is required and when the larger linear address space enabled by 64-bit addressing is not required.

The test software for both the 32-bit and 64-bit memory addressing models under Windows 7 on Hosts A and C was compiled and run under 64-bit Windows 7. A compiler switch was used to specify the target of either 32-bit or 64-bit memory addressing. In contrast, the results for Linux were achieved by compiling and running the test software separately on all three hosts (A, B, and C), first under the 32-bit version of Linux and then under the 64-bit version.

Effects of Sizes of Array Elements and Indexes on Runtimes

The use of an 8-byte `long int` as the array index results in a slight decrease in runtime but only under the 32-bit memory model with GCC. The additional use of 8-byte `long int` as the type for the two-dimensional array, in conjunction with an 8-byte `long int` array index, may contribute to a reduced runtime.

Effects of Specification of `register` Storage Class on Runtimes

Specifying that array indexes are in the `register` storage class reduces runtimes for the unoptimized code but has essentially no effect for optimized code. Although the use of the `register` storage class is deprecated in C++11, it still may be used in C and also in C++ when there is a clear need. This specification may be useful for achieving faster processing during debugging if a debugger that precludes compiler optimization is in use. Because the specification has no effect on code optimized by the compiler, it is unlikely to be useful in production code.

Specifying that array indexes are in the `register` storage class suggests to the compiler that it generate code to keep array addresses in processor registers rather than in RAM. Performing address arithmetic using indexes and offsets maintained in processor registers during loop processing is much faster than repeatedly fetching and storing indexes and offsets from and to variables in RAM.

Combination of Factor Levels Leading to the Shortest Runtimes

Analysis of the data shows that the speed of processing of two-dimensional arrays with code created by C or C++ compilers is variable. Data from this comparative study of two-dimensional array processing show that a suite of test programs (all programs performing the same amount of computational processing of data in two-dimensional arrays) produced runtimes ranging from a low of about 7 seconds up to a high of about 160 seconds.

The following combination of factor levels produced the shortest runtimes on Host A:

- Linux operating system with GCC;
- No randomization;
- C language;
- Coding technique 3;
- 8-byte `long int` array elements;[11]
- 8-byte `long int` array indexes;
- 32-bit code memory model; and
- Compiler optimization.

On Host B, the combination is the same, except the language leading to the shortest runtimes is C++.

On Host C, the combination includes C (like Host A and unlike Host B) but differs from both Host A and Host B—the array elements and array indexes are 4-byte `int`'s and the operating system and compiler are Windows 7 and Visual Studio 2010.

Practical Advice for Software Developers

Based on the findings in the preceding section, the following points and principles are suggested to C and C++ programmers for developing efficient software for processing raster-graphic images or other two-dimensional arrays:

1. Test code with several combinations of different levels of the 10 factors listed in the preceding section.
2. Run tests on the target system intended to host the production code (rather than on a separate development host) since processing speed is influenced by processor architecture and operating system.
3. Use compiler optimizations for production code (and consider all available optimizations).
4. Use coding technique 3 unless timing tests show that another technique is faster if the slight increase in the use of RAM entailed by this technique is acceptable in the application. (Please refer to the "Standard and Alternative Coding Techniques" subsection for a description of the three coding techniques for array processing.)
5. Consider running production code compiled with 32-bit memory addressing.
6. Experiment with varying the type (`int` or `long int`) of integer data arrays (if feasible) or the type of the array indexes.
7. Consider comparing a C version of the created code with a C++ version if possible.
8. Compare code created with different compilers if possible.
9. Specify that array indexes are of `register` storage class when using a debugger.
10. Analyze the code loops used to process large arrays and search for ways to enable the compiler to predict the value of each array index in the next iteration of the loop.

[11]Both compilers used in the comparative study compile code with `sizeof(int)` equal to 4 bytes and `sizeof(long int)` equal to 8 bytes. Not all compilers generate code with these same sizes for `int`s and `long int`s.

Conclusions and Recommendations

This report provides specific, technical advice to C or C++ programmers for reducing the serial runtime for computational loops used to process two-dimensional arrays. The advice is based on a comparative study that evaluated the effects of ten different factors on runtimes for two-dimensional arrays. The findings and advice in this report were motivated by the performance requirements of land-change modeling, but they can also be applied to the processing of raster-graphic images and other two-dimensional arrays in computer modeling and scientific computation.

At the outset of this study, the choice of coding technique was expected to be one of the strongest determinants of array-processing speed, after compiler optimizations. This expectation, however, was not borne out by the study. The following is a summary of conclusions.

Array-processing runtimes can vary by a factor of 10 or more, depending on choices of the levels of the factors that determine processing speed. Therefore, programmers can substantially reduce runtimes for their array-processing code by carefully selecting features of both the runtime environment and their C or C++ source code. Because the combination of choices leading to the fastest processing is difficult to predict, the best way to achieve fast processing is by conducting test runs on potential target hosts for various combinations of processing-speed determinants.

The study left several important questions unanswered, and to help answer those questions, four follow-up actions are recommended:

Recommendation 1—Test Standard Template Library Containers

Runtimes should be tested using the `array`, `valarray`, and `vector` containers from the C++ standard template library.

Recommendation 2—Test with More Compilers and Optimizations

Tests should be run with more compilers: the C language family of LLVM compilers for Linux, and the Intel C++ Compilers for Windows (and perhaps for Linux). Additionally, a test should be run by compiling Fortran array-processing code into an object module and then linking it into a C or C++ program.[12] Additional compiler optimization options should also be tested.

Recommendation 3—Test Additional Data Types

Tests should be run with arrays of floating-point numbers and other data types.

Recommendation 4—Build a Software Suite for Optimizing Array Processing

The study establishes that no simple prescription exists for fast and efficient two-dimensional array processing in C and C++. There is still a need for fast and efficient array-processing software, and, therefore, programmers still need help developing this software. A suite of software that facilitates the rapid testing of array-processing software for any particular host with a variety of compilers should be developed. This suite of software would automatically generate and test array-processing code for as many factor levels as possible. Such software, given an array-processing loop, would configure, compile, and run timing tests for numerous coding techniques, data types for array elements, array-index data types, address models, programming languages, and compiler optimization levels.

[12]Since C and C++ arrays are stored in memory in row-major sequence, and Fortran arrays are stored in column-major sequence, an appropriate transformation is required when linking a Fortran-compiled object module into a C or C++ program.

References Cited

Donato, D.I., 2017, Runtimes for tests of array-processing speed: U.S. Geological Survey data release, https://doi.org/10.5066/F7W66HZS.

Intel, 2016, Intel C++ Compilers: Intel developer zone web page, accessed May 14, 2016, at https://software.intel.com/en-us/c-compilers.

Josuttis, N.M., 2012, The C++ standard library: A tutorial and reference (2d ed.): Upper Saddle River, N.J., Addison-Wesley, 1,099 p.

McConnell, Steve, 2004, Code complete (2d ed.): Redmond, Wash., Microsoft Press, 914 p.

Myers, Scott, 2014, Effective modern C++—42 specific ways to improve your use of C++11 and C++14: Sebastopol, Calif., O'Reilly Media Inc., 315 p.

Stroustrup, Bjarne, 2013, The C++ programming language (4th ed.): Upper Saddle River, N.J., Addison-Wesley, 1,346 p.

University of Illinois LLVM Project, 2016, clang—A C language family frontend for LLVM: Clang project web page, accessed May 14, 2016, at http://clang.llvm.org.

Appendix 1. Scatter Diagrams

This appendix contains 40 scatter diagrams illustrating the effects of the two factors from the comparative factorial study that have the greatest effects on runtimes. These two factors—compiler optimization and randomization—are shown in diagrams for each of 20 combinations of operating environment, as follows—

1. Host—A, B, or C
2. Operating system and code address model—32-bit or 64-bit addressing on 64-bit Windows 7; 32-bit Linux; or 64-bit Linux
3. Language—C or C++

The 40 diagrams are organized into 20 pairs of diagrams, with one pair for each of the 20 combinations making up an operating environment. The pairing of diagrams allows for visual comparison of the effects of compiler optimization and randomization within the same operating environment. Comparison between two operating environments is facilitated by use of the same scale for runtime, in seconds, in all 40 diagrams. Each diagram also includes a trend line for visual reference.

Each diagram for C language runtimes displays up to 28 scatter points; and each diagram for C++ language runtimes displays up to 20 scatter points. Most diagrams, however, show fewer scatter points than these maxima because their respective underlying data include several scatter points with very nearly equal runtimes on both the vertical and horizontal axes. For simplicity, the diagrams use the same symbol for both single and multiple scatter points.

When the points on a diagram occur in two clusters (as seen, for example, in fig. 1–9), the effects of factors other than optimization or randomization in this operating environment are comparatively weak. By contrast, when the points on a diagram are widely scattered (as seen, for example, in fig. 1–20), the effects of factors other than optimization and randomization are strong.

For operating environments with short runtimes, the points are confined to the lower left of the diagram (as seen, for example, in fig. 1–26). By contrast, for operating environments with longer runtimes, the points are found farther to the right and extending farther upward (as seen, for example, in figs. 1–23 and 1–24).

Although runtimes cannot be directly compared between hosts, because of the differences in processing due to hardware differences, the patterns of scatter may be compared both within and between hosts.

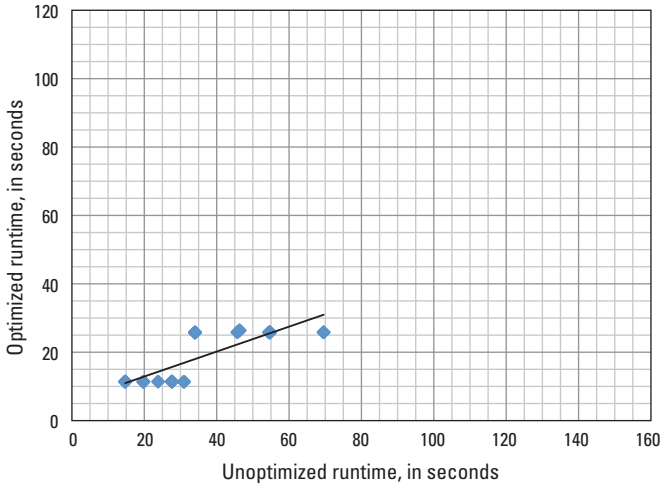

Figure 1–1. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—32-bit Linux—C language.

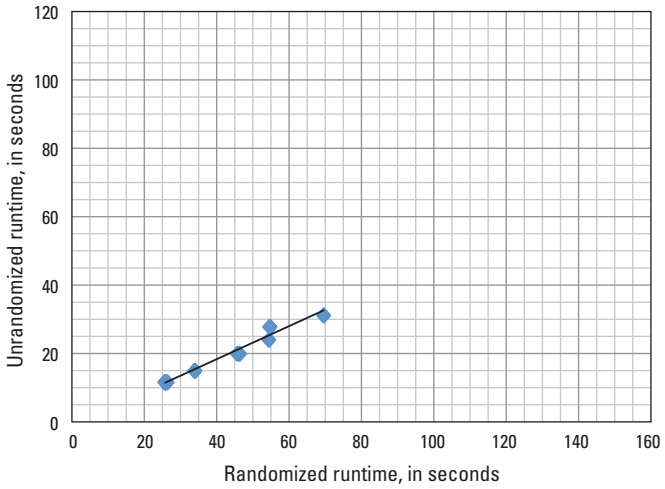

Figure 1–2. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—32-bit Linux—C language.

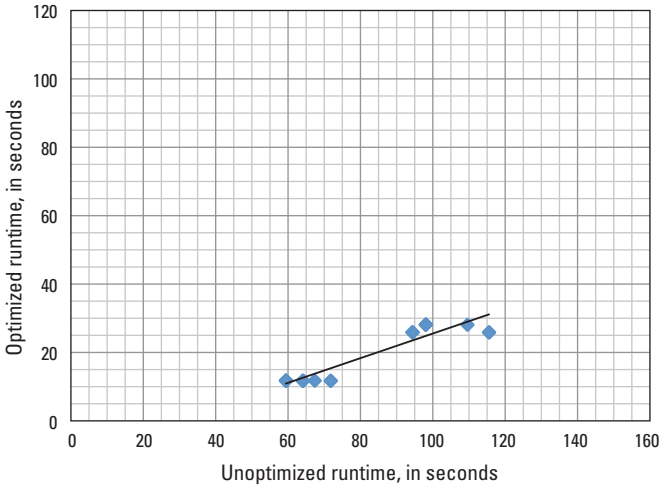

Figure 1–3. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—32-bit Linux—C++ language.

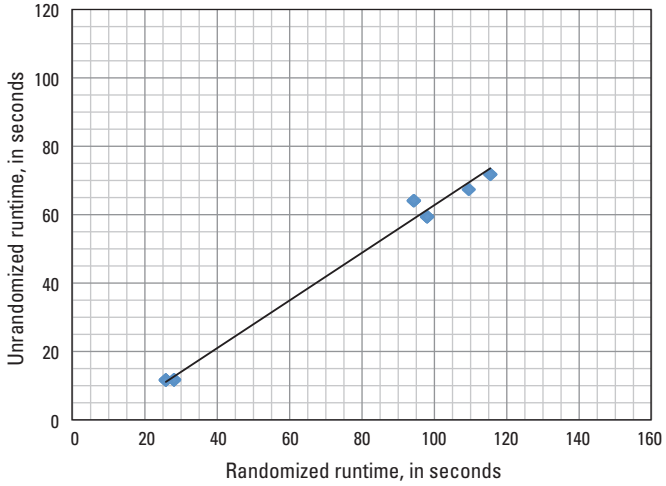

Figure 1–4. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—32-bit Linux—C++ language.

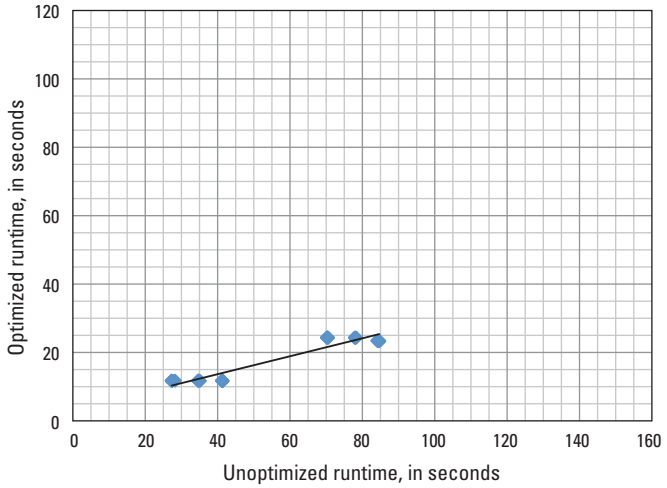

Figure 1–5. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—32-bit Windows—C language.

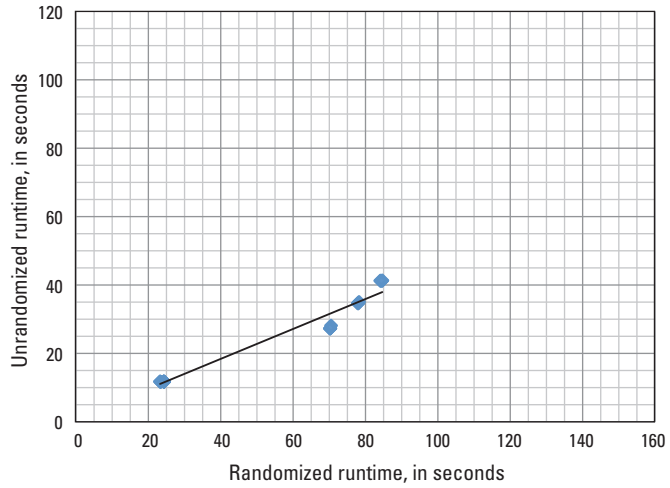

Figure 1–6. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—32-bit Windows—C language.

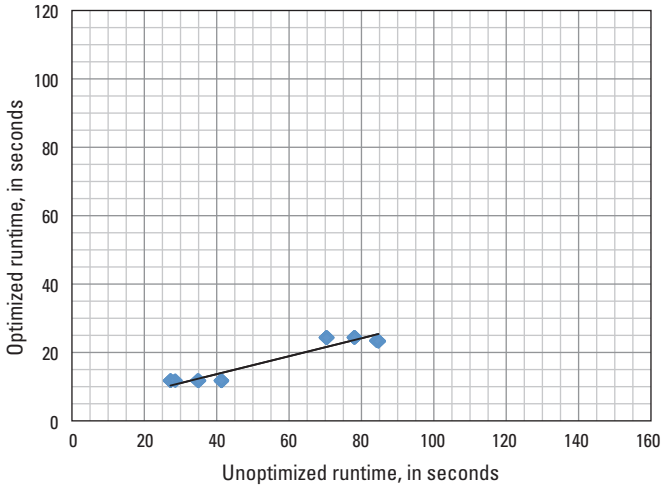

Figure 1–7. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—64-bit Windows—C language.

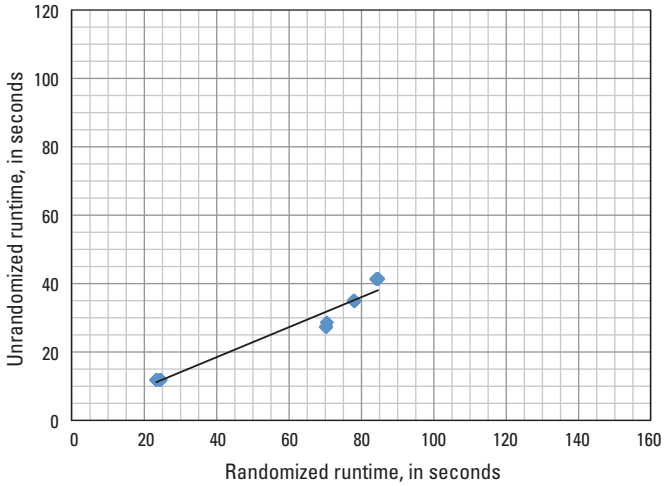

Figure 1–8. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—64-bit Windows—C language.

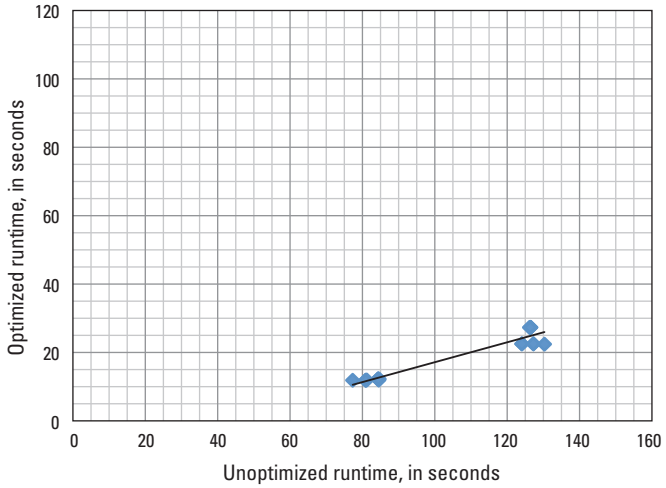

Figure 1–9. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—32-bit Windows—C++ language.

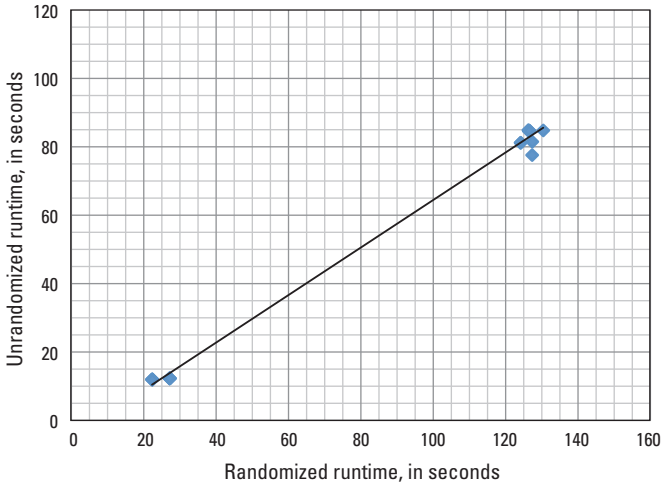

Figure 1–10. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—32-bit Windows—C++ language.

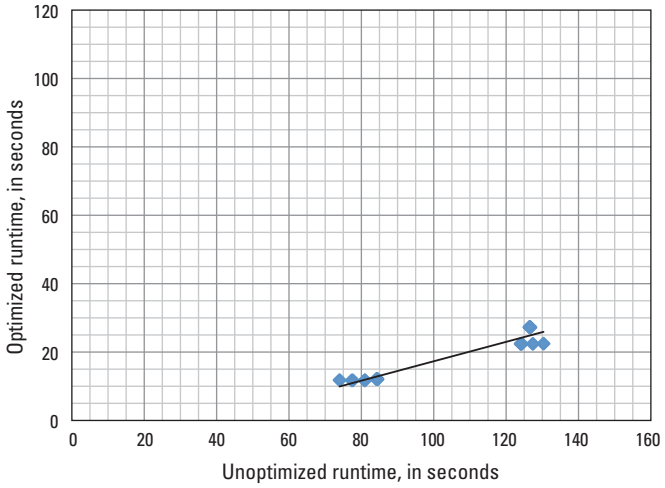

Figure 1–11. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—64-bit Windows—C++ language.

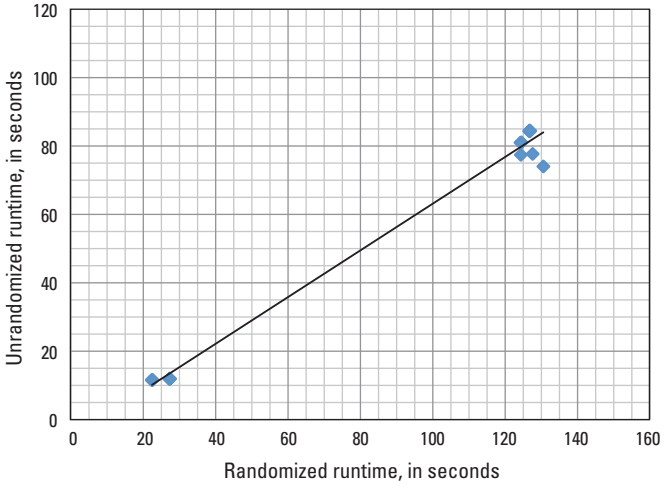

Figure 1–12. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—64-bit Windows—C++ language.

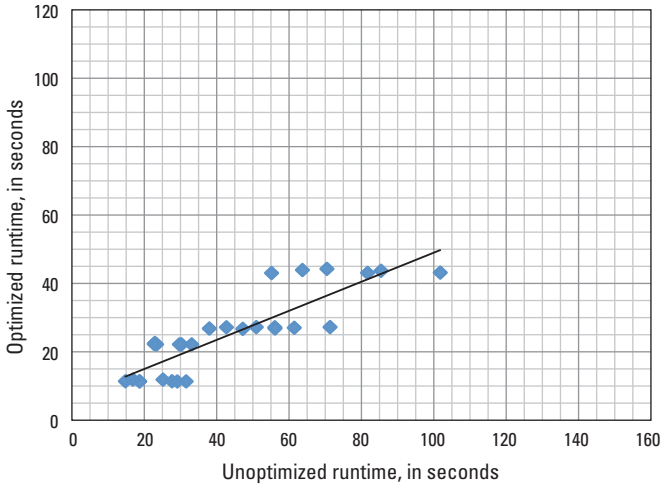

Figure 1–13. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—64-bit Linux—C language.

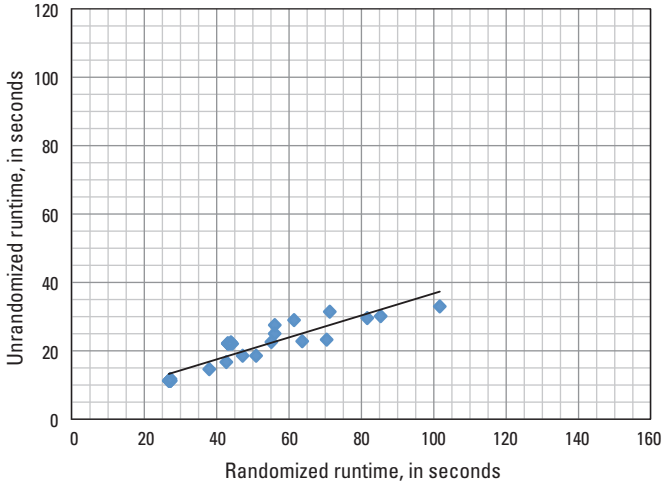

Figure 1–14. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—64-bit Linux—C language.

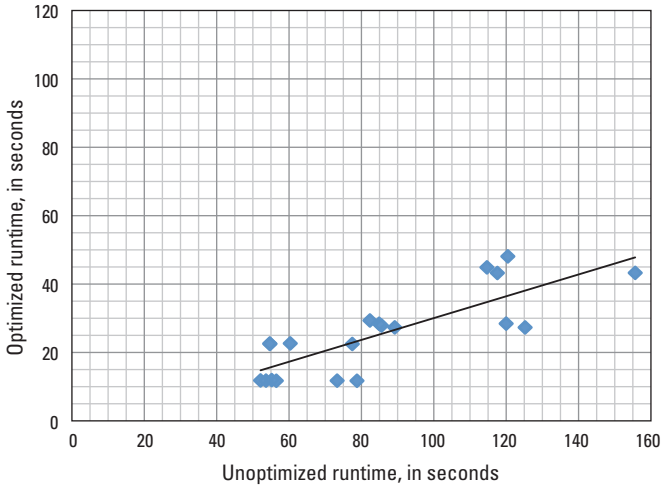

Figure 1–15. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host A—64-bit Linux—C++ language.

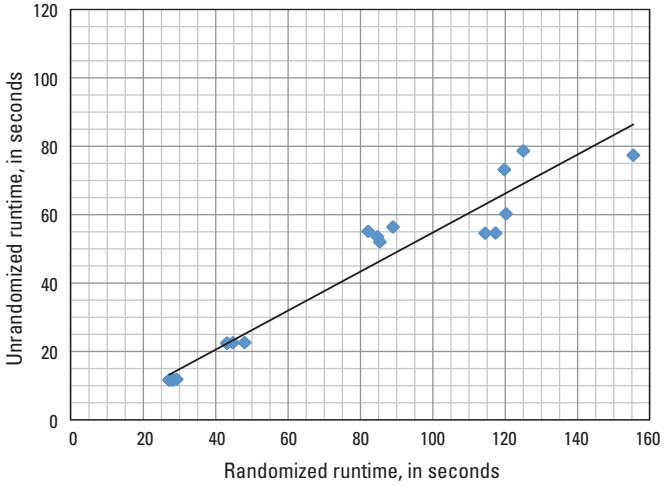

Figure 1–16. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host A—64-bit Linux—C++ language.

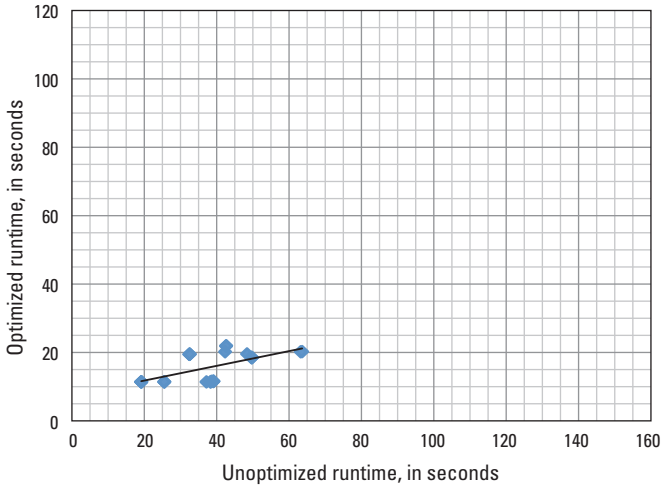

Figure 1–17. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host B—32-bit Linux—C language.

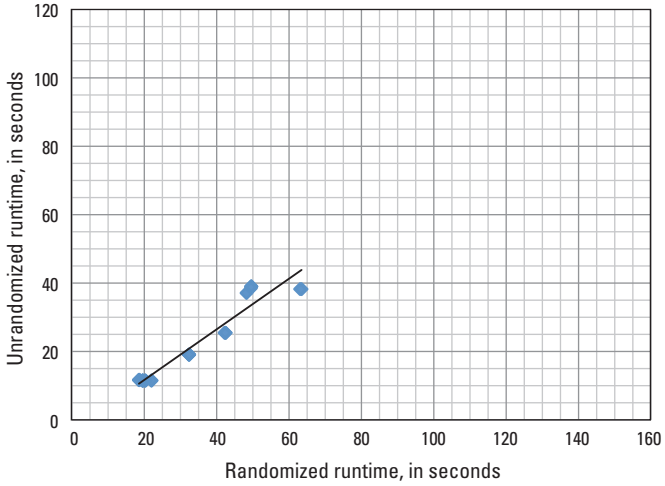

Figure 1–18. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host B—32-bit Linux—C language.

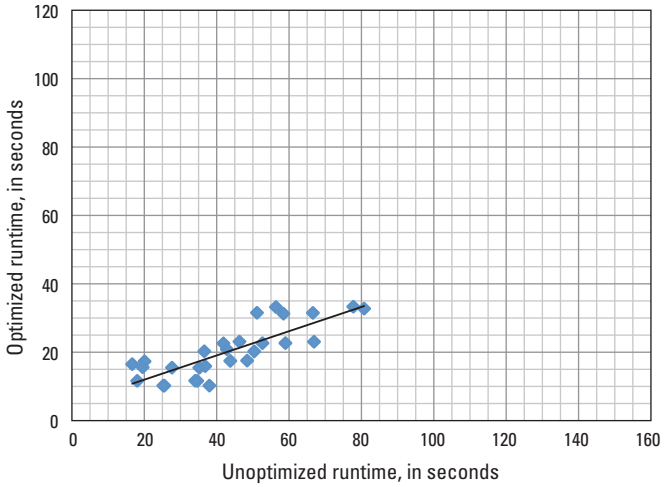

Figure 1–19. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host B—64-bit Linux—C language.

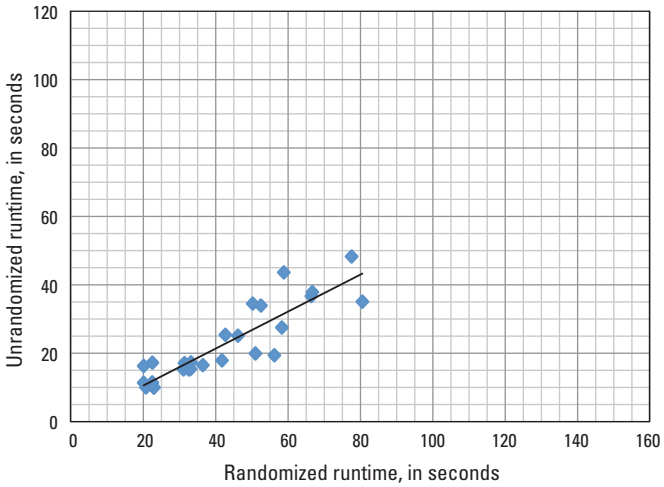

Figure 1–20. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host B—64-bit Linux—C language.

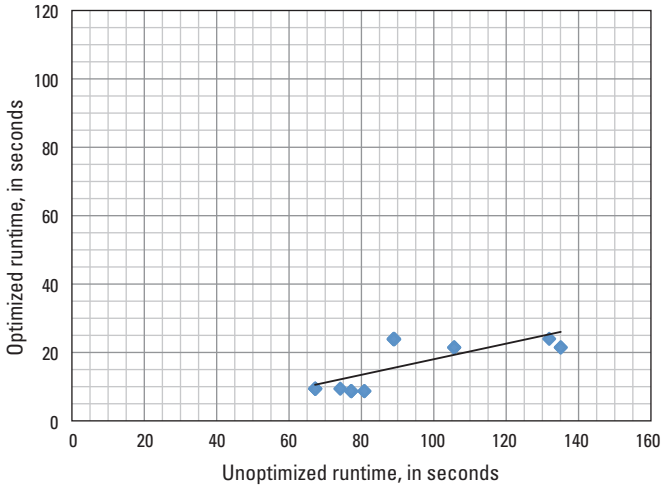

Figure 1–21. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host B—32-bit Linux—C++ language.

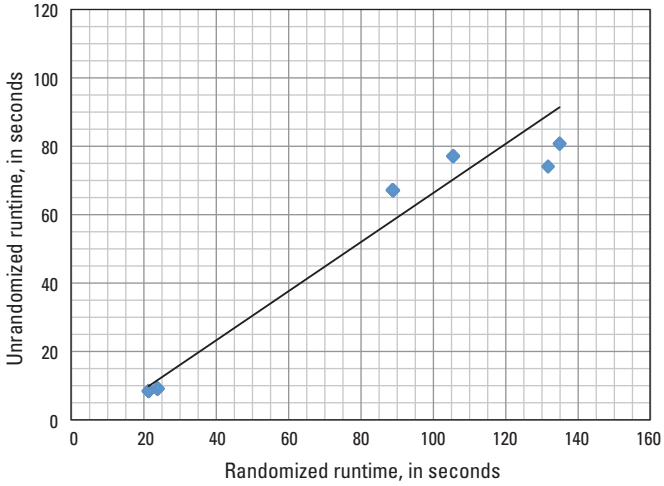

Figure 1–22. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host B—32-bit Linux—C++ language.

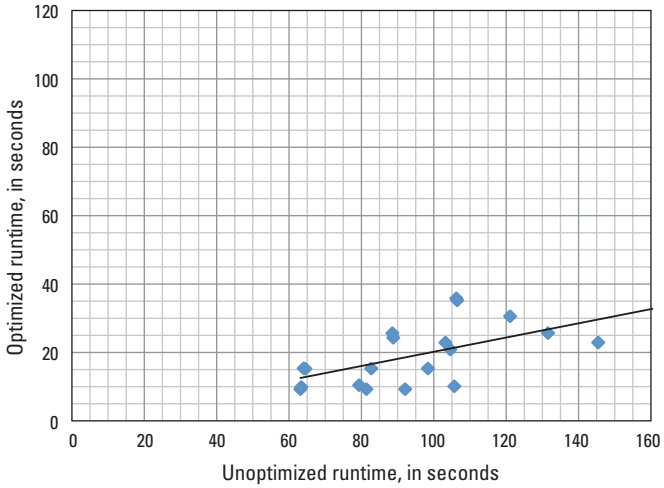

Figure 1–23. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host B—64-bit Linux—C++ language.

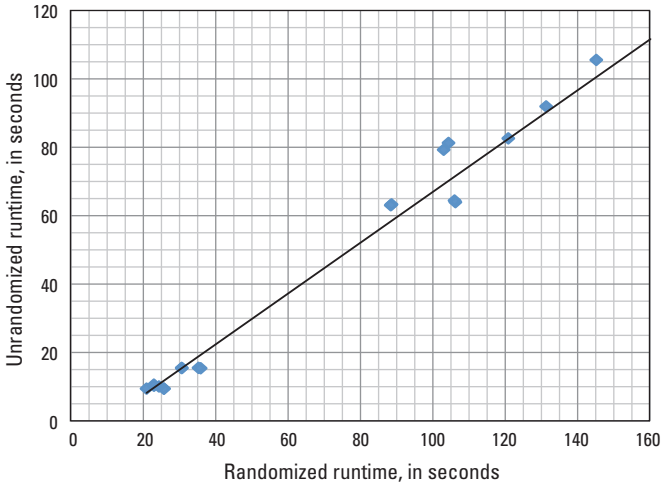

Figure 1–24. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host B—64-bit Linux—C++ language.

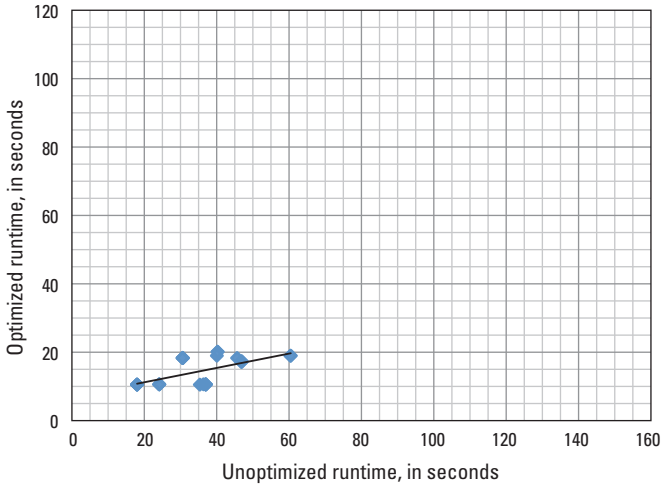

Figure 1–25. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—32-bit Linux—C language.

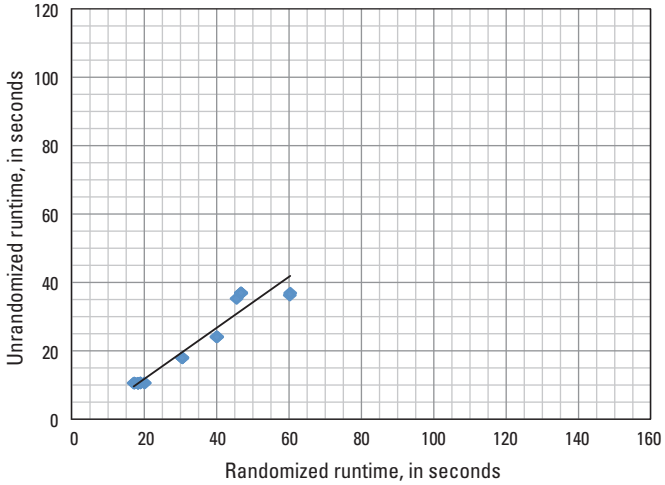

Figure 1–26. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—32-bit Linux—C language.

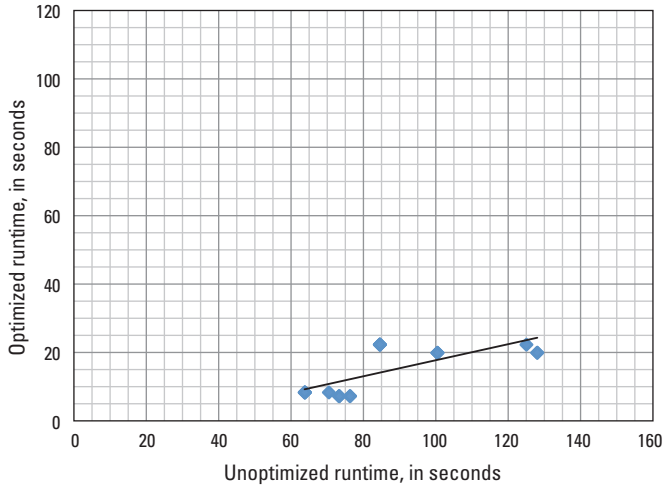

Figure 1–27. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—32-bit Linux—C++ language.

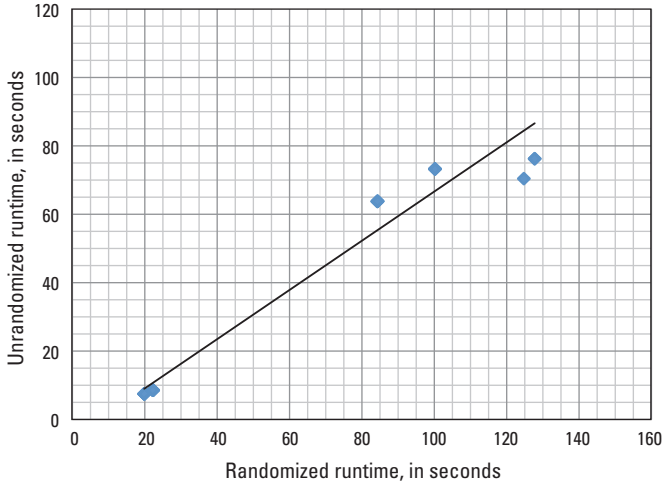

Figure 1–28. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—32-bit Linux—C++ language.

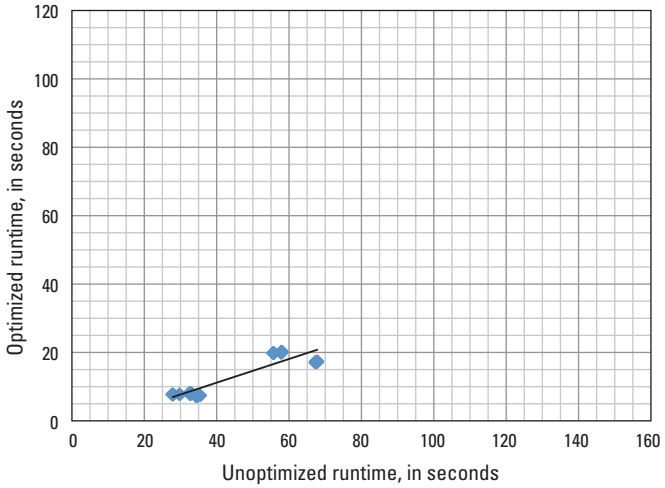

Figure 1–29. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—32-bit Windows—C language.

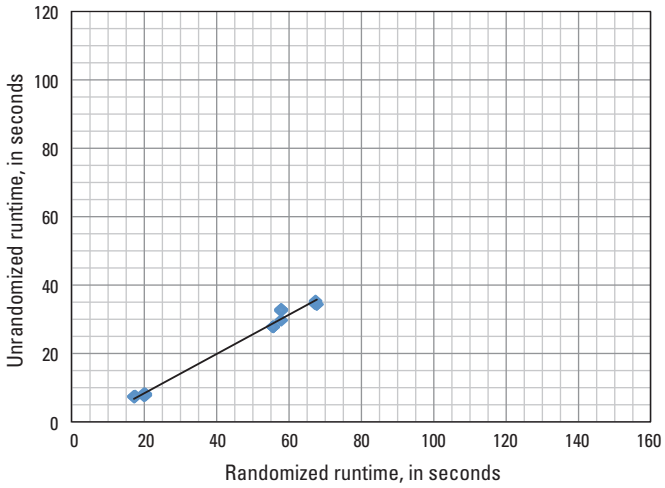

Figure 1–30. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—32-bit Windows—C language.

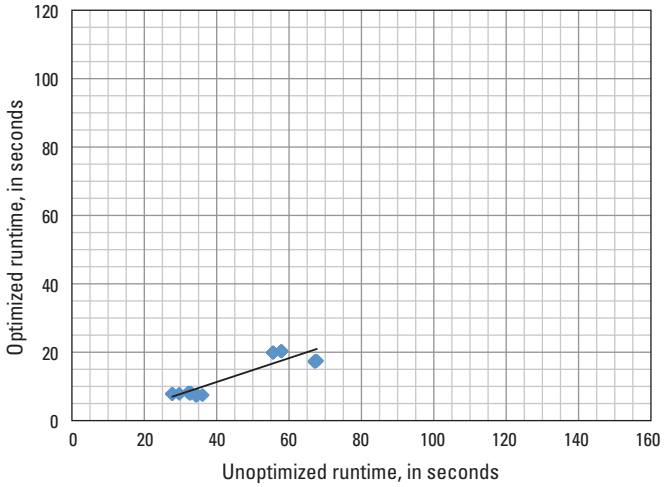

Figure 1–31. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—64-bit Windows—C language.

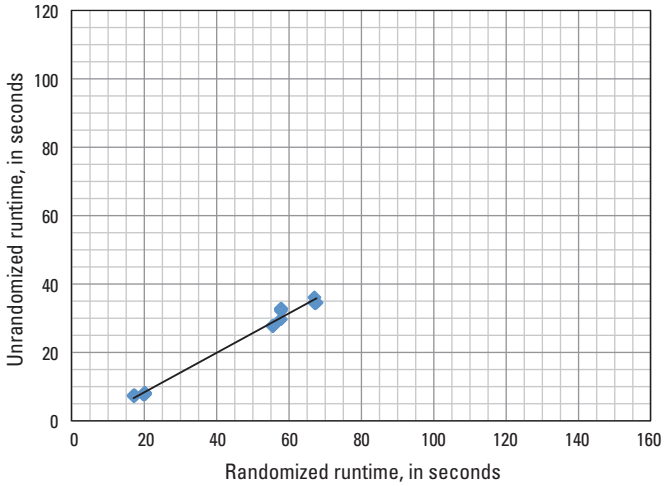

Figure 1–32. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—64-bit Windows—C language.

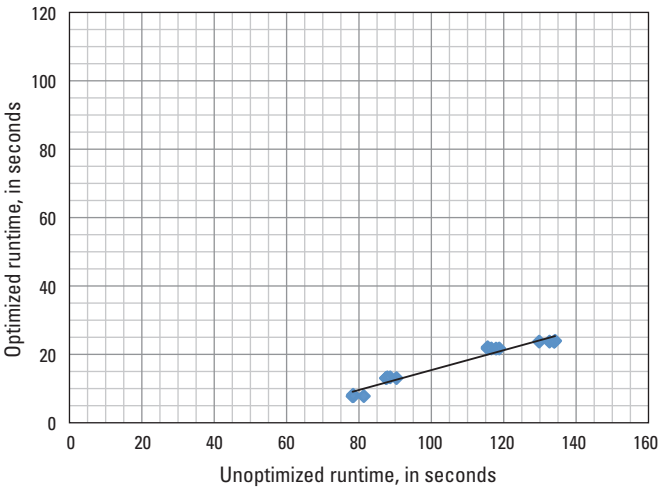

Figure 1–33. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—32-bit Windows—C++ language.

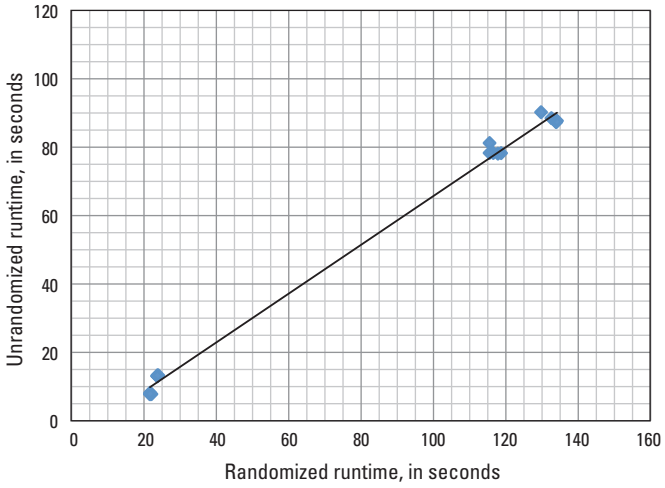

Figure 1–34. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—32-bit Windows—C++ language.

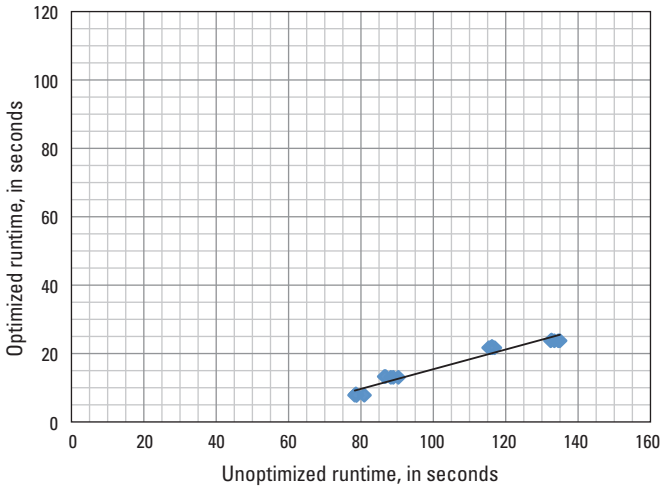

Figure 1–35. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—64-bit Windows—C++ language.

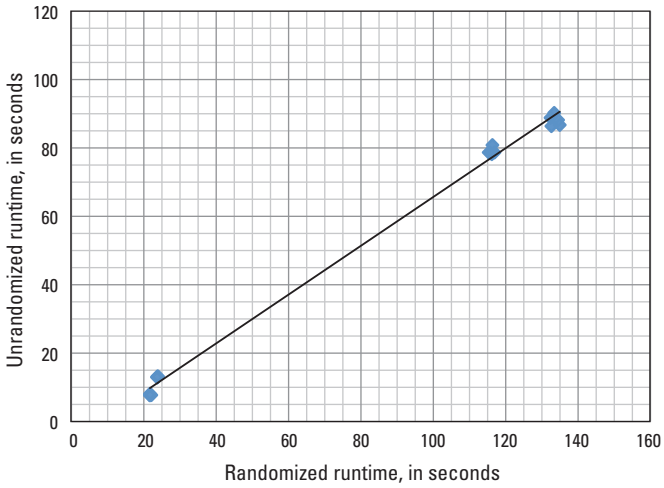

Figure 1–36. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—64-bit Windows—C++ language.

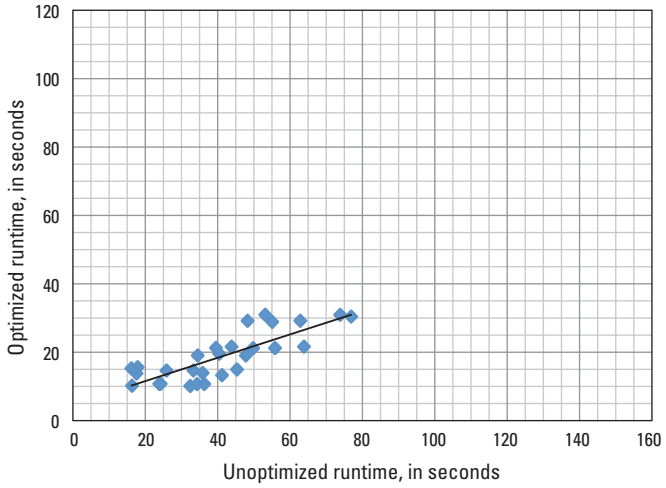

Figure 1–37. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—64-bit Linux—C language.

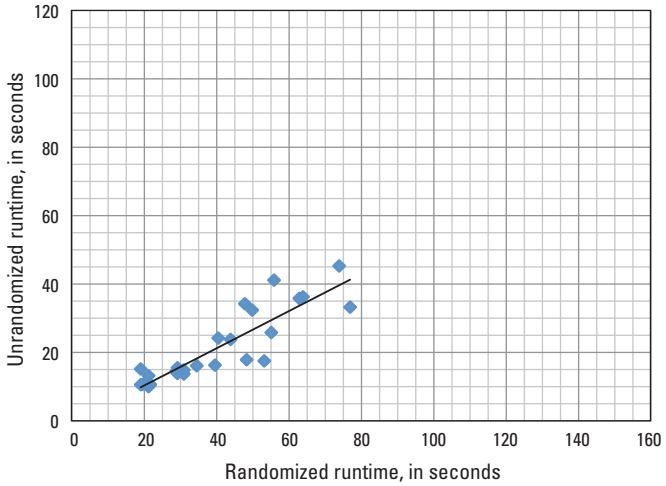

Figure 1–38. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—64-bit Linux—C language.

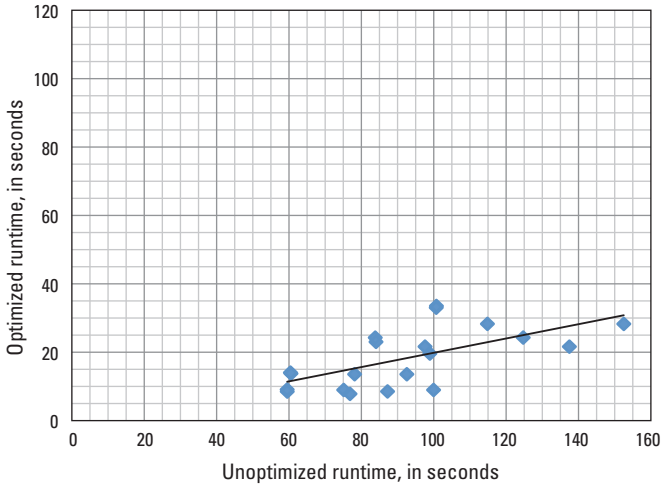

Figure 1–39. Scatter diagram of unoptimized runtimes versus optimized runtimes for Host C—64-bit Linux—C++ language.

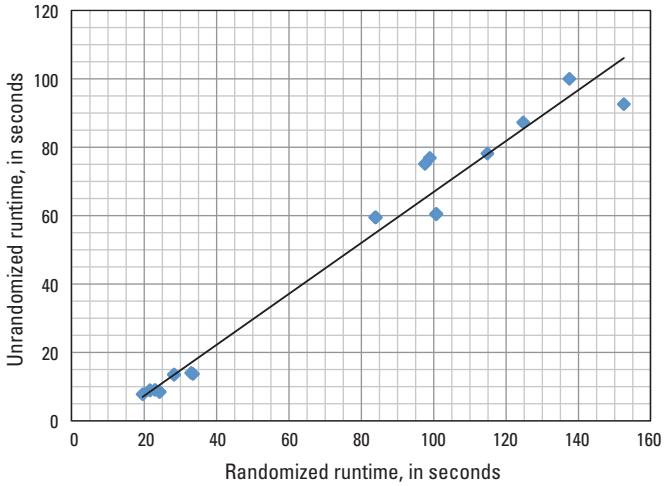

Figure 1–40. Scatter diagram of randomized runtimes versus unrandomized runtimes for Host C—64-bit Linux—C++ language.

Appendix 2. Boxplots

This appendix contains 20 figures, one for each combination of factors making up an operating environment. (These operating environments are the same as in appendix 1.) Each figure contains a cluster of boxplots, with 12 boxplots in each figure for operating environments using the C programming language, and with eight boxplots in each figure for operating environments using the C++ programming language. The main purpose of these figures is to allow easy comparison of the three coding techniques.

Each figure is divided into four quadrants:

1. Randomized and optimized test runs
2. Randomized and unoptimized test runs
3. Unrandomized and optimized test runs
4. Unrandomized and unoptimized test runs

Within each quadrant is a boxplot for coding techniques 1, 2, and 3 in figures that involve C, and there is a boxplot for coding techniques 2 and 3 in figures that involve C++. Coding technique 1 was not tested with C++.

Each boxplot shows the observed values of a processing-time index value for the operating environment. The index is the quotient of an actual processor runtime in seconds divided by the longest runtime in seconds observed for the operating environment. The index value can range from a low value, such as 0.1, to a maximum value of 1.0. Low values correspond to short runtimes for efficient processing; high values correspond to long runtimes for inefficient processing. The following sample shows the conceptual layout of each boxplot within the figures.

This sample shows four processing-time index values, indicating that factors other than optimization or randomization have a pronounced effect on relative runtimes.

Because the same scale is used for all of the boxplots, the numerical labels for the vertical axes of the boxplots are omitted for simplicity.

Each boxplot for C language runtimes represents 28 index values; and each boxplot for C++ language runtimes represents 20 index values. None of the boxplots show this many index values, however, because their respective sets of index values include many very nearly equal indexes. For simplicity, the boxplots use the same symbol for both single and multiple index values.

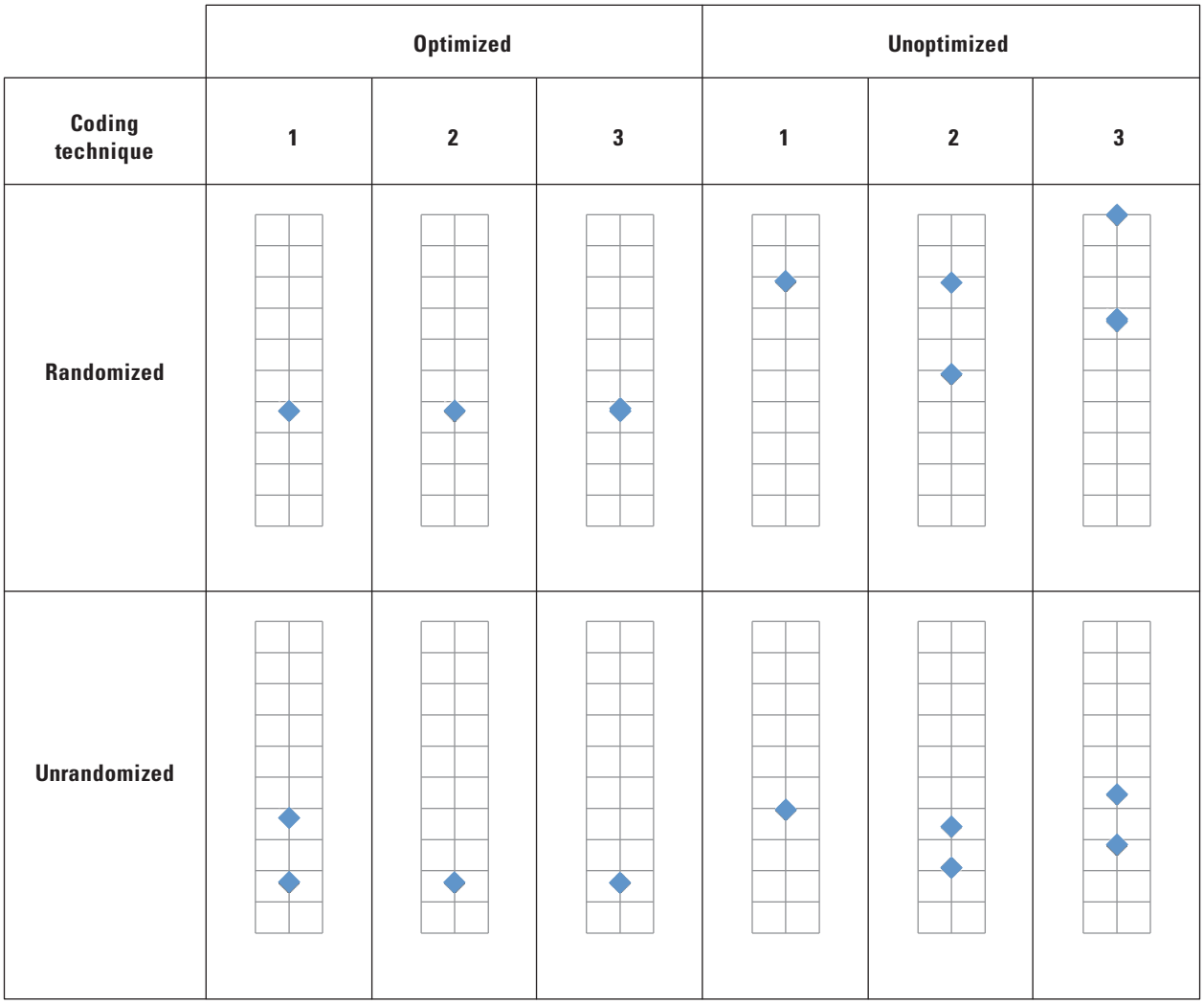

Figure 2–1. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host A—32-bit Linux—C language.

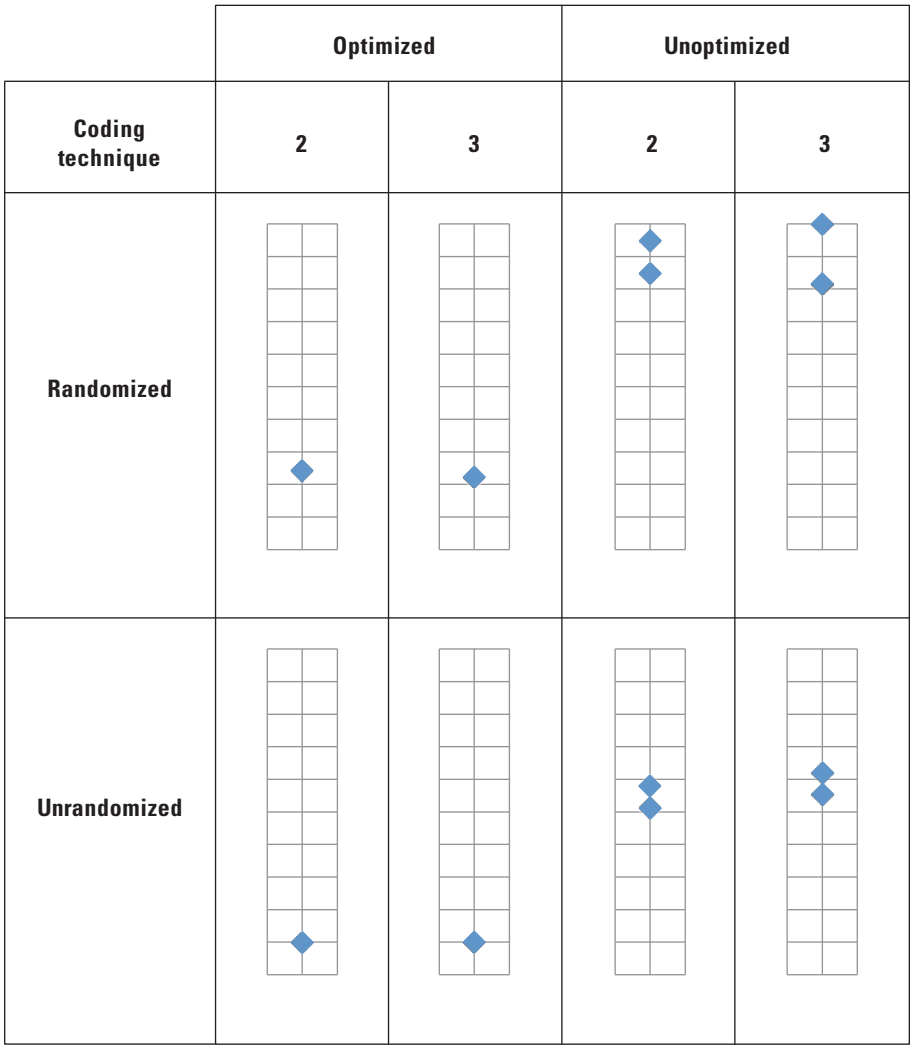

Figure 2–2. Boxplots of relative runtimes with coding techniques 2 and 3 on Host A—32-bit Linux—C++ language.

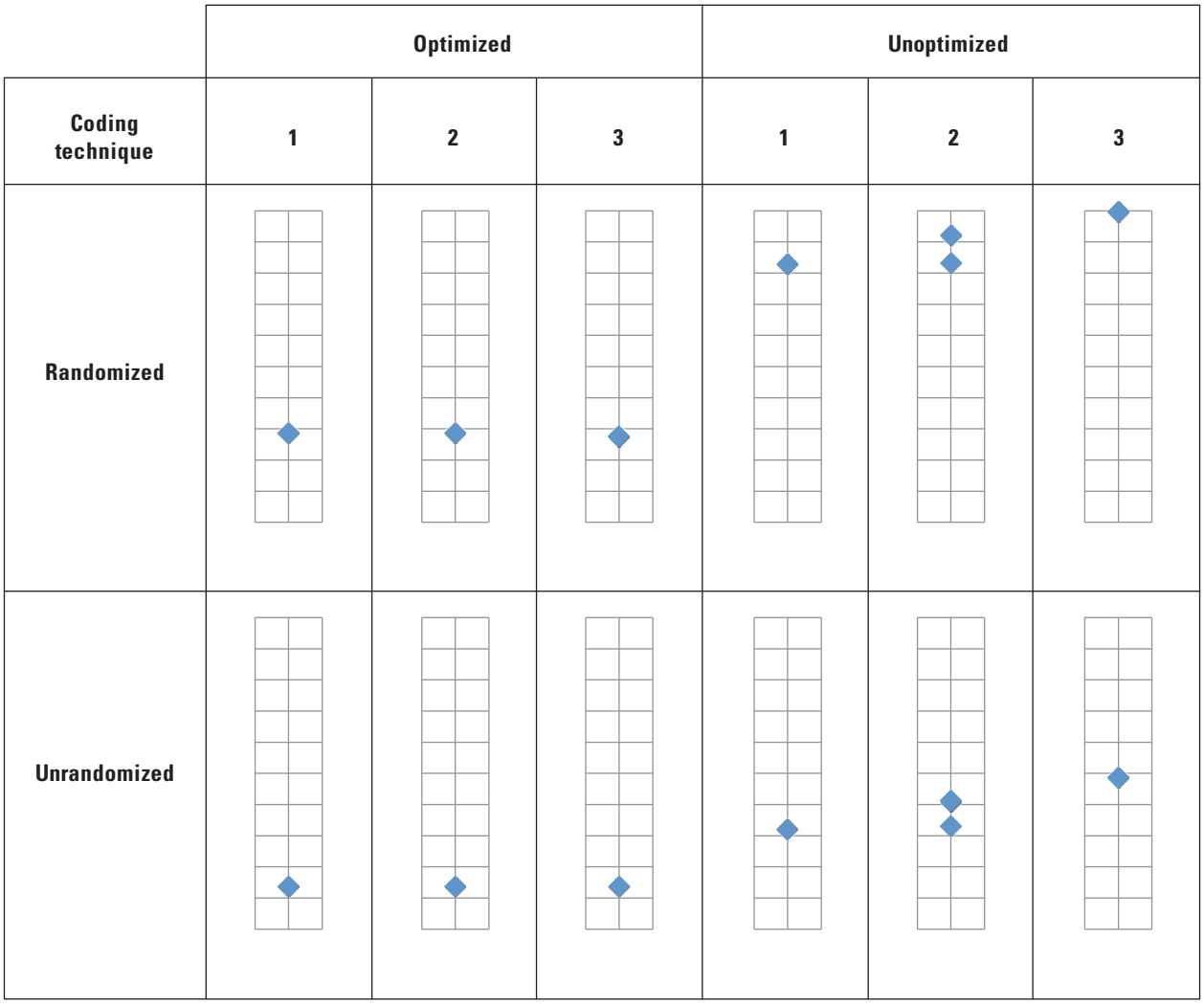

Figure 2–3. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host A—32-bit Windows—C language.

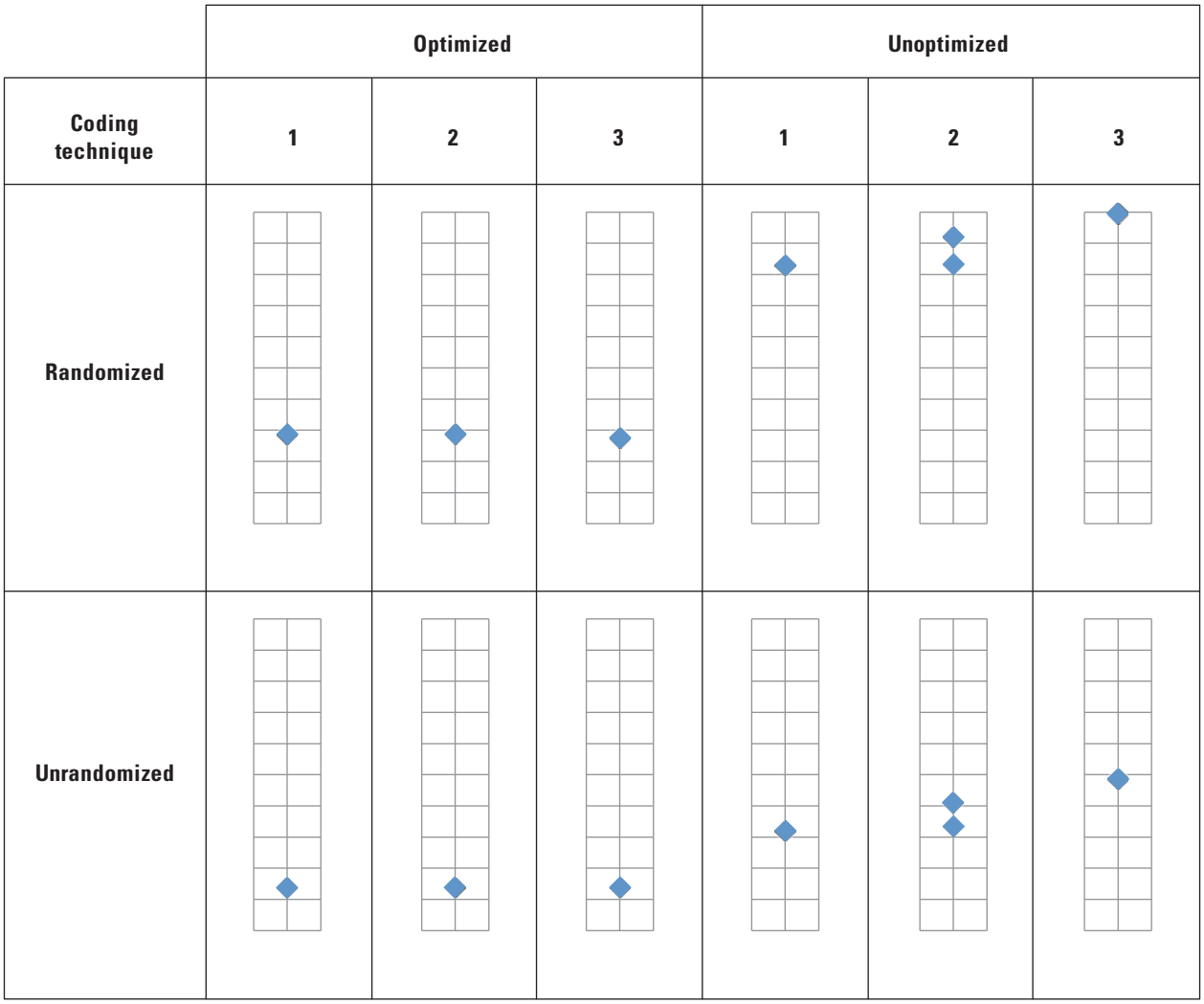

Figure 2–4. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host A—64-bit Windows—C language.

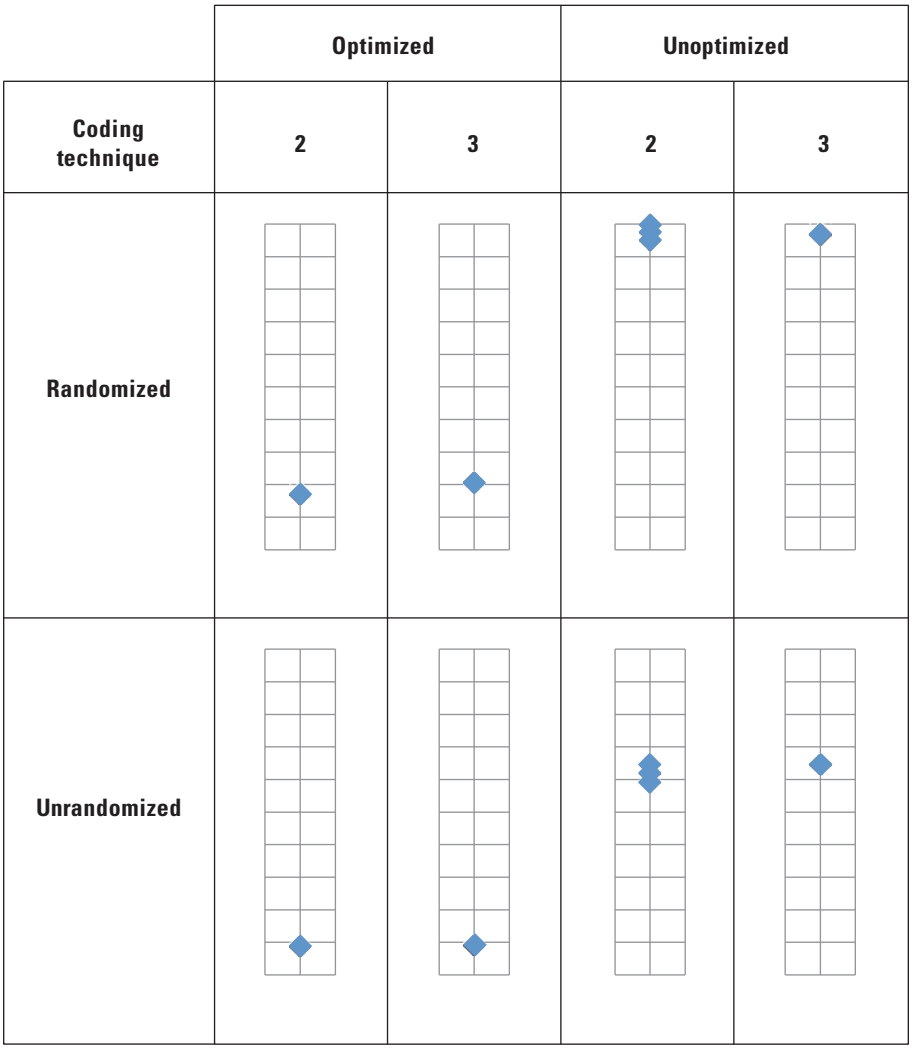

Figure 2–5. Boxplots of relative runtimes with coding techniques 2 and 3 on Host A—32-bit Windows—C++ language.

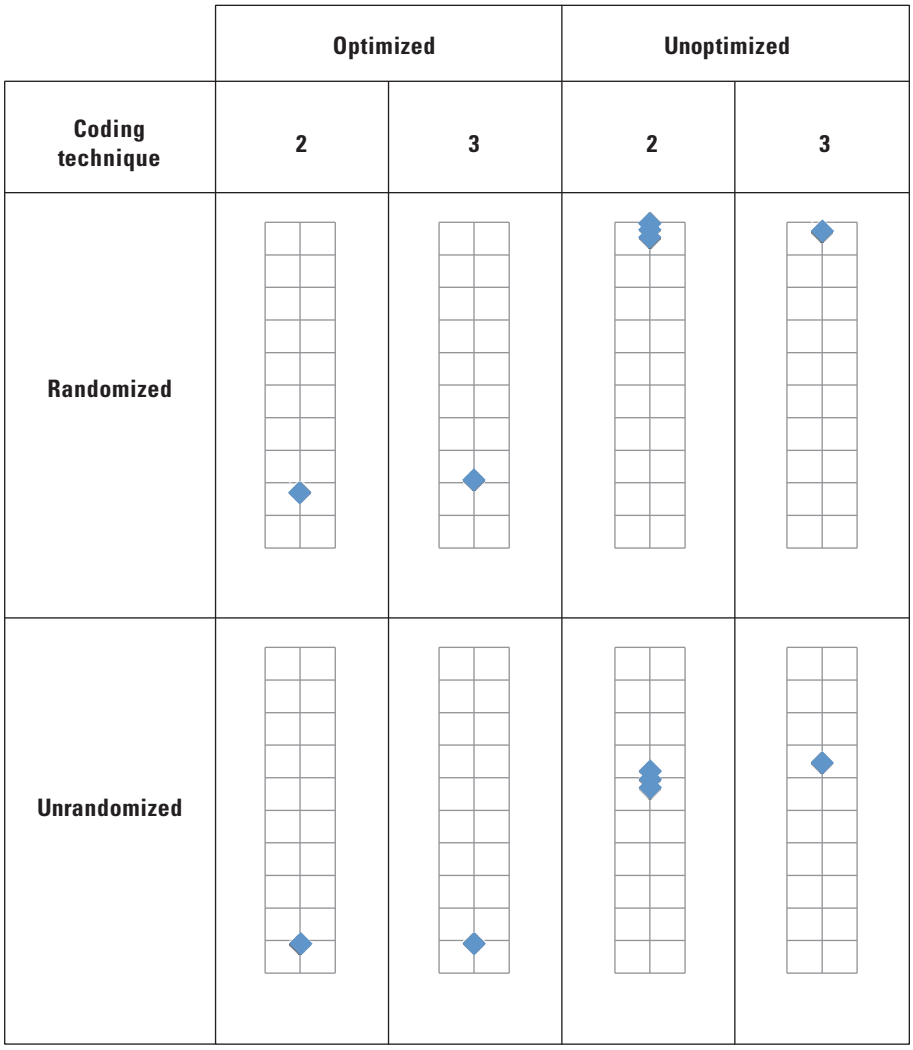

Figure 2–6. Boxplots of relative runtimes with coding techniques 2 and 3 on Host A—64-bit Windows—C++ language.

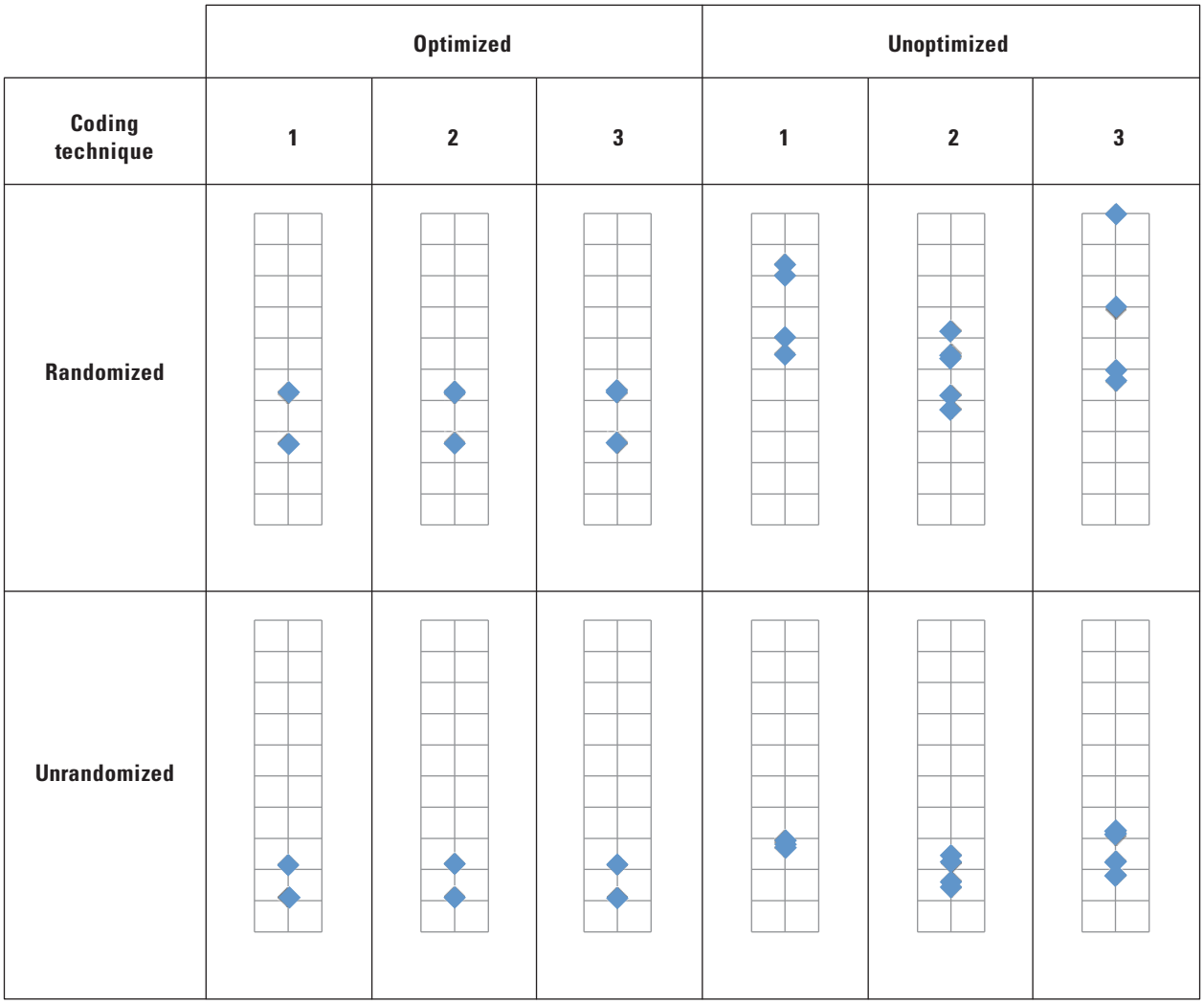

Figure 2–7. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host A—64-bit Linux—C language.

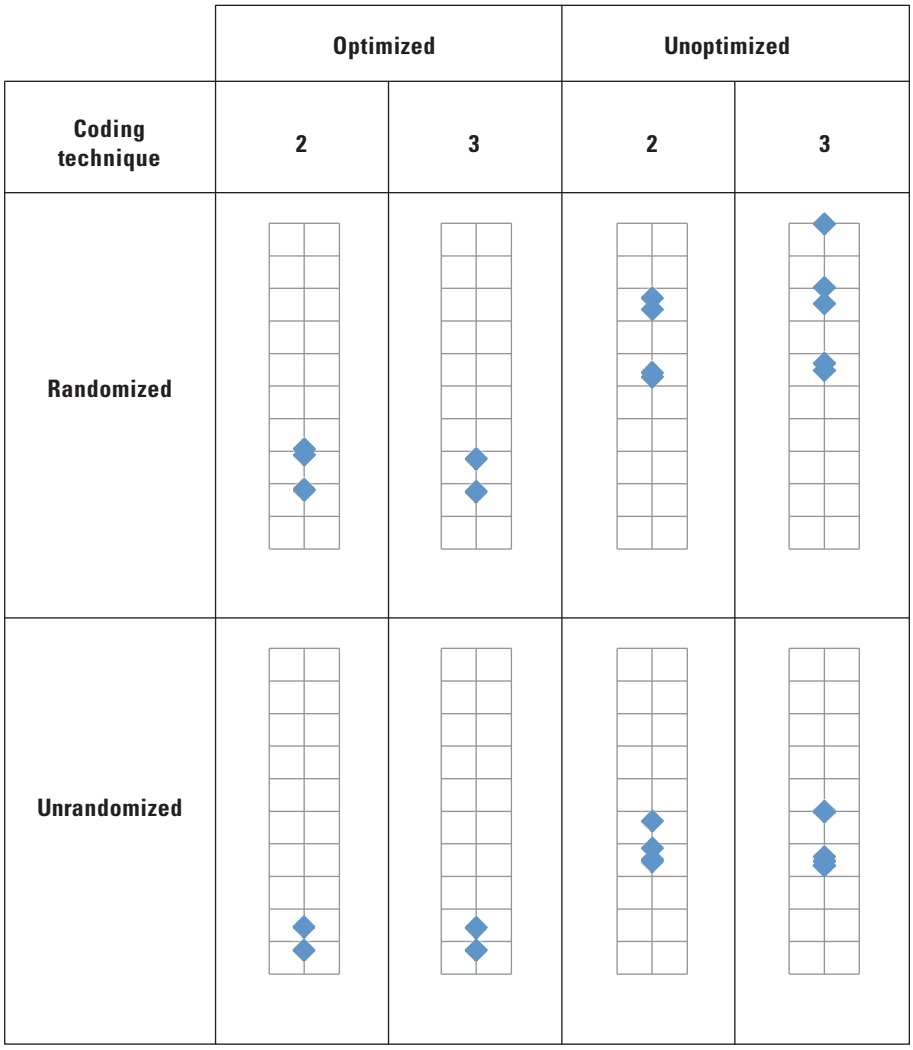

Figure 2–8. Boxplots of relative runtimes with coding techniques 2 and 3 on Host A—64-bit Linux—C++ language.

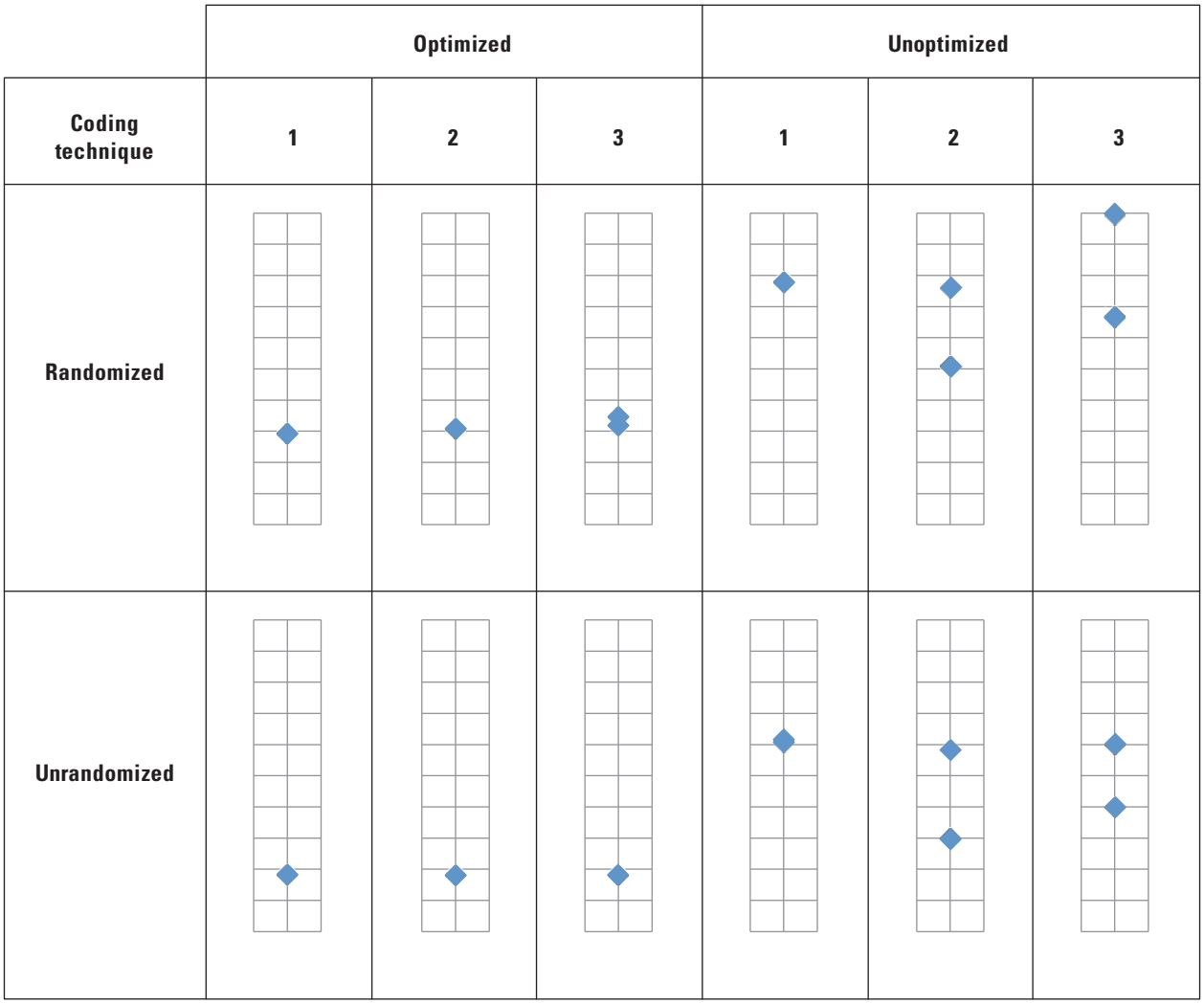

Figure 2–9. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host B—32-bit Linux—C language.

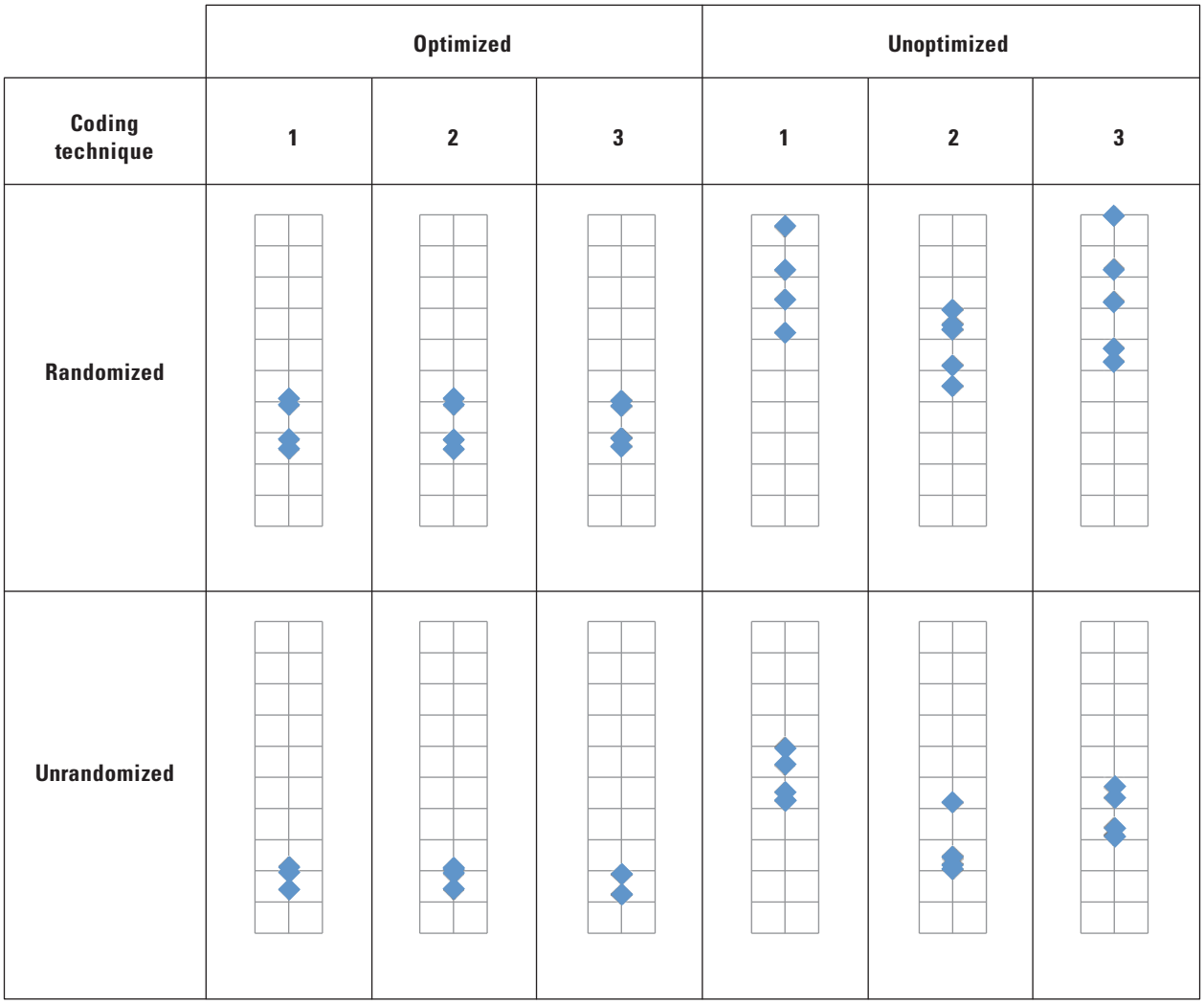

Figure 2–10 Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host B—64-bit Linux—C language.

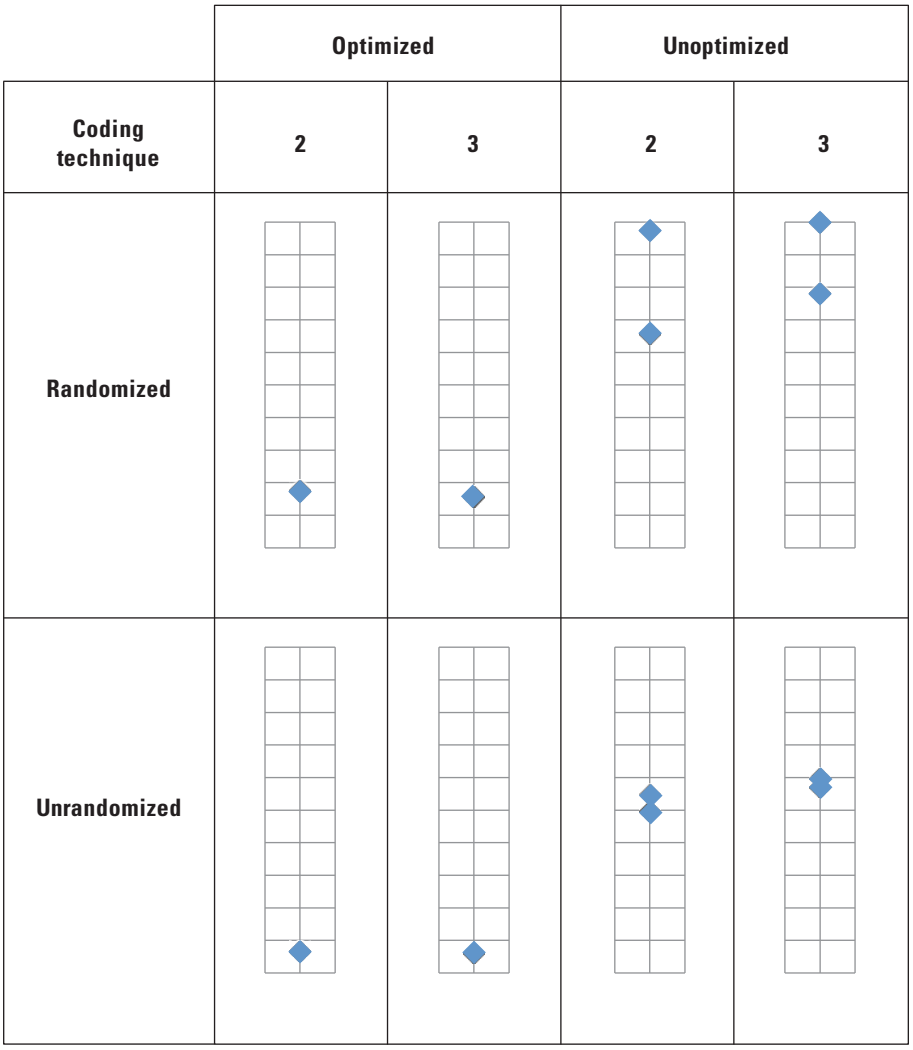

Figure 2–11. Boxplots of relative runtimes with coding techniques 2 and 3 on Host B—32-bit Linux—C++ language.

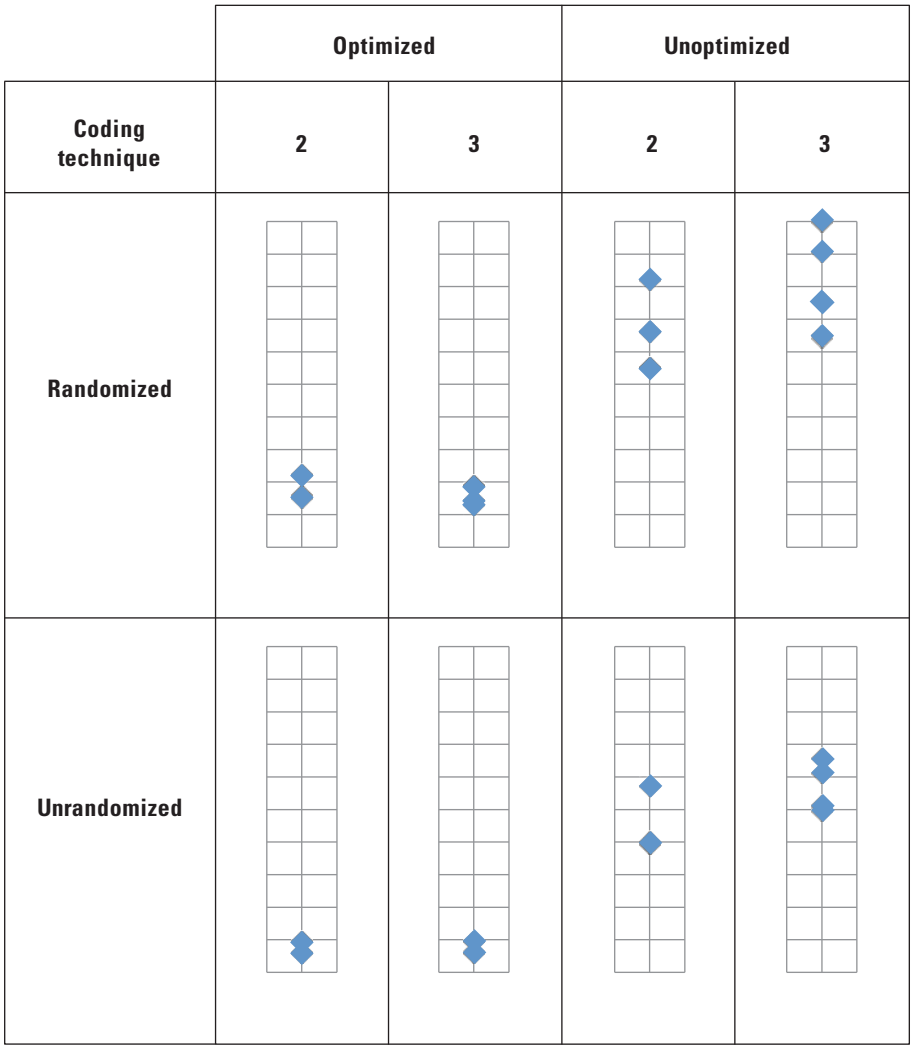

Figure 2–12 Boxplots of relative runtimes with coding techniques 2 and 3 on Host B—64-bit Linux—C++ language.

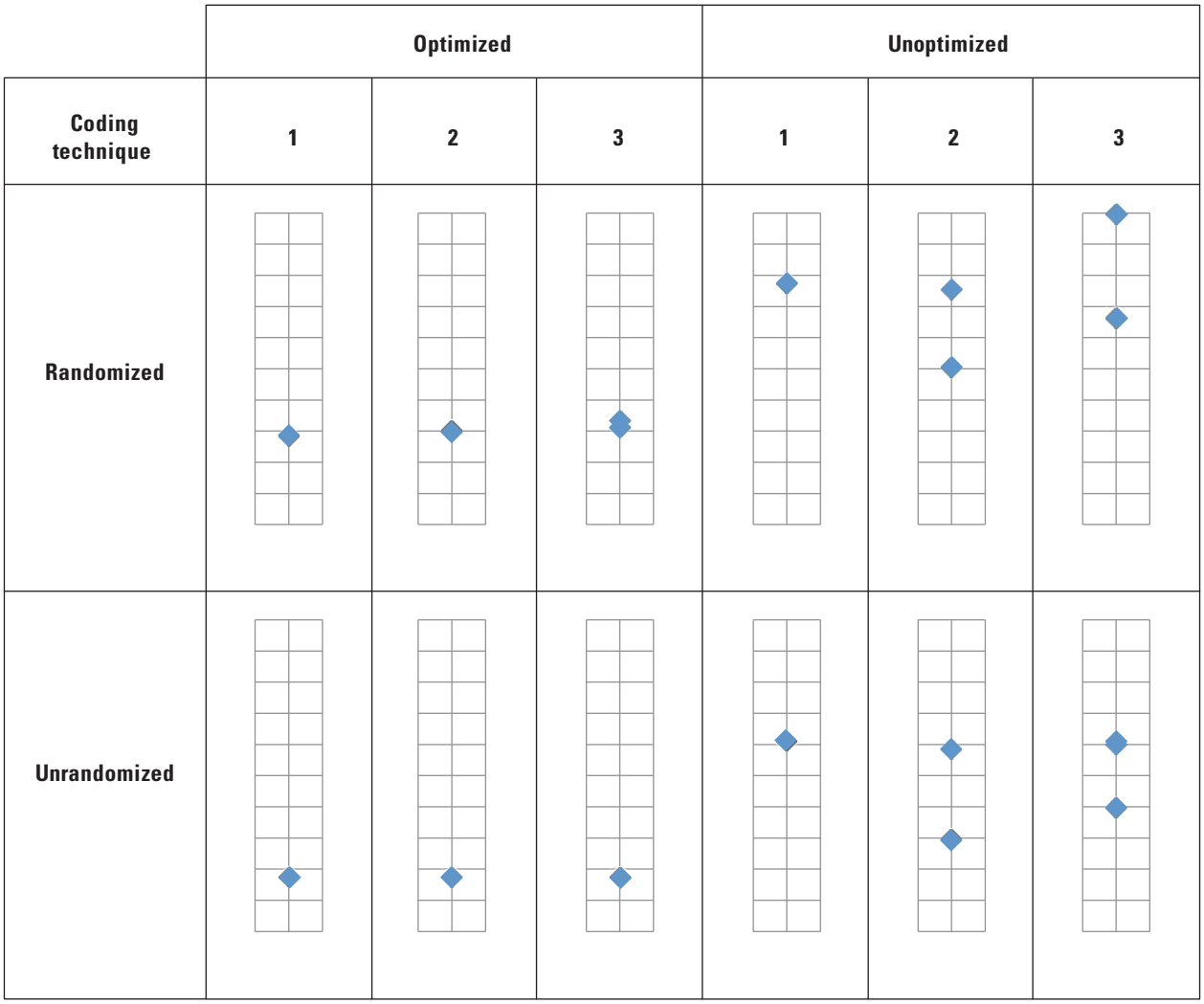

Figure 2–13. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host C—32-bit Linux—C language.

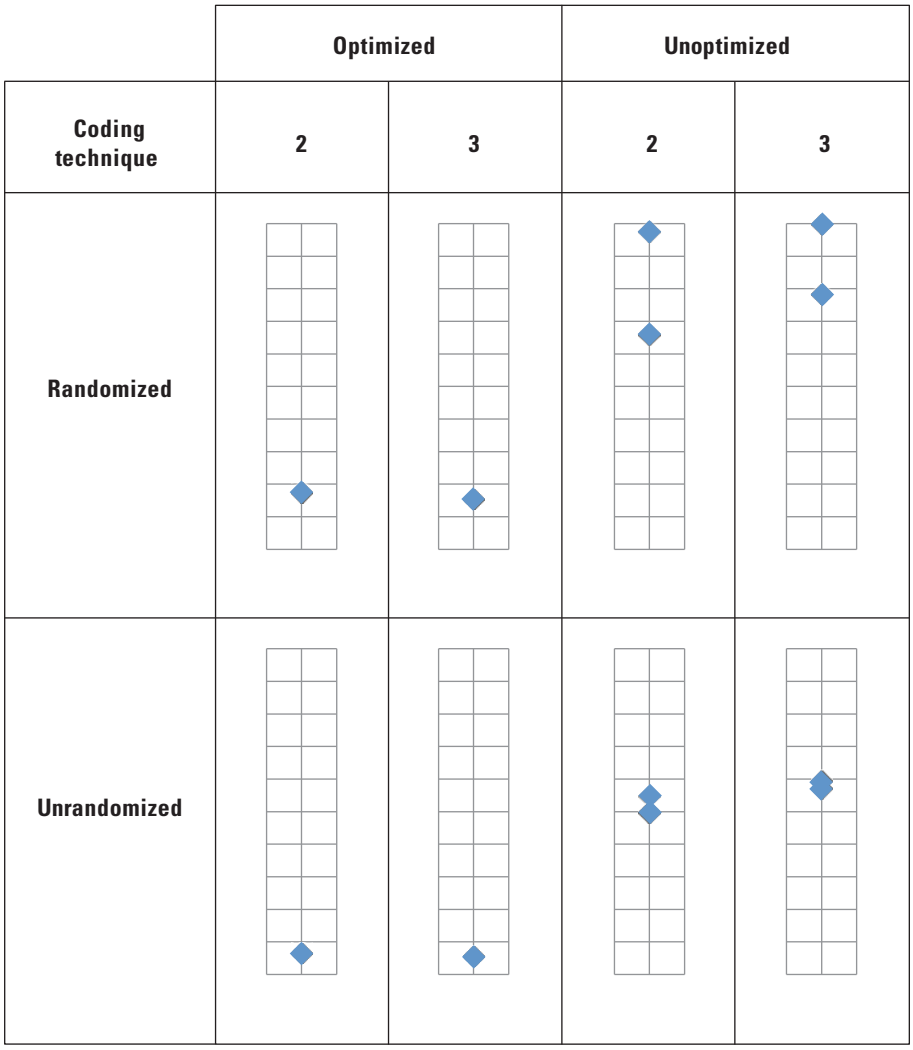

Figure 2–14 Boxplots of relative runtimes with coding techniques 2 and 3 on Host C—32-bit Linux—C++ language.

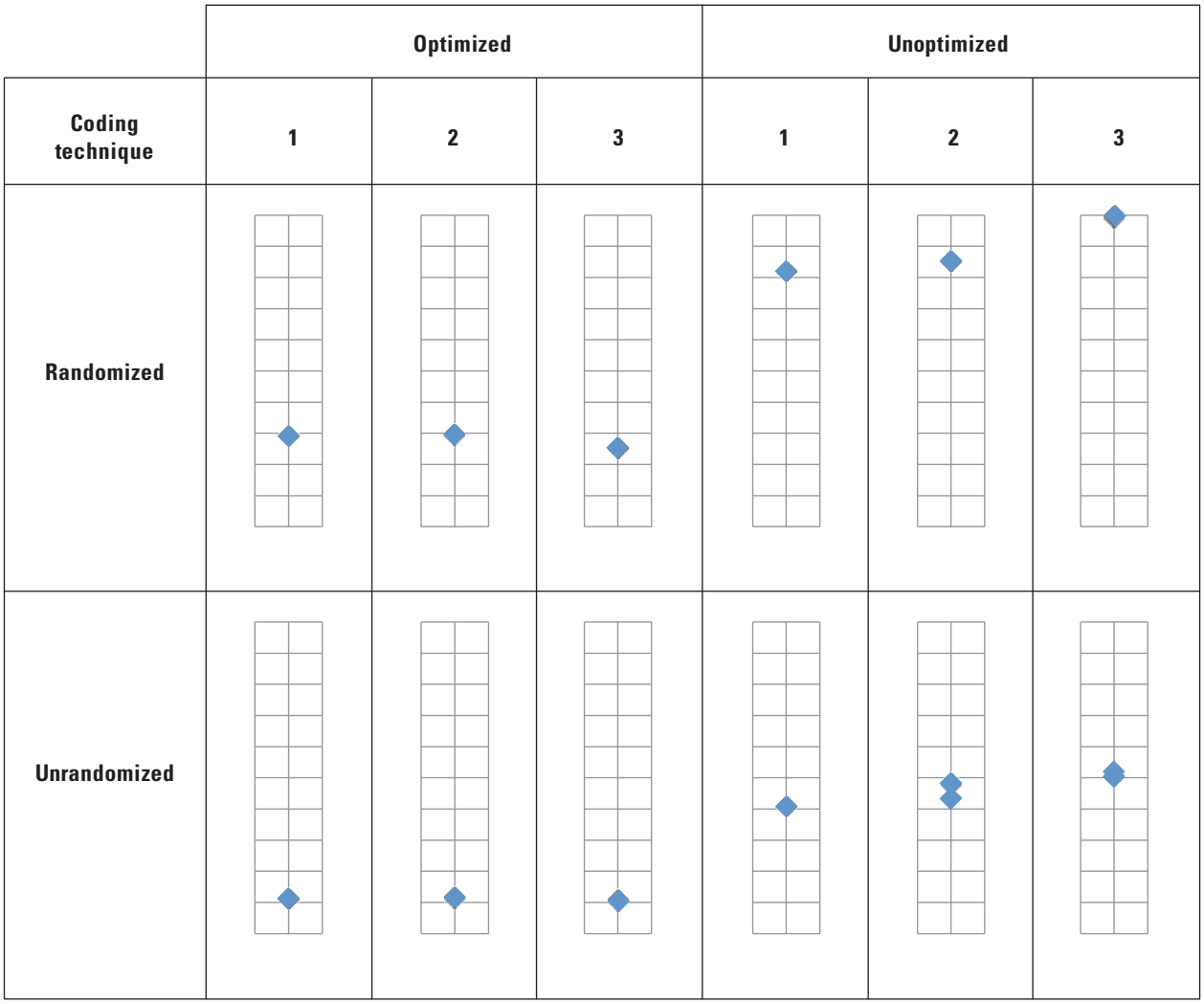

Figure 2–15. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host C—32-bit Windows—C language.

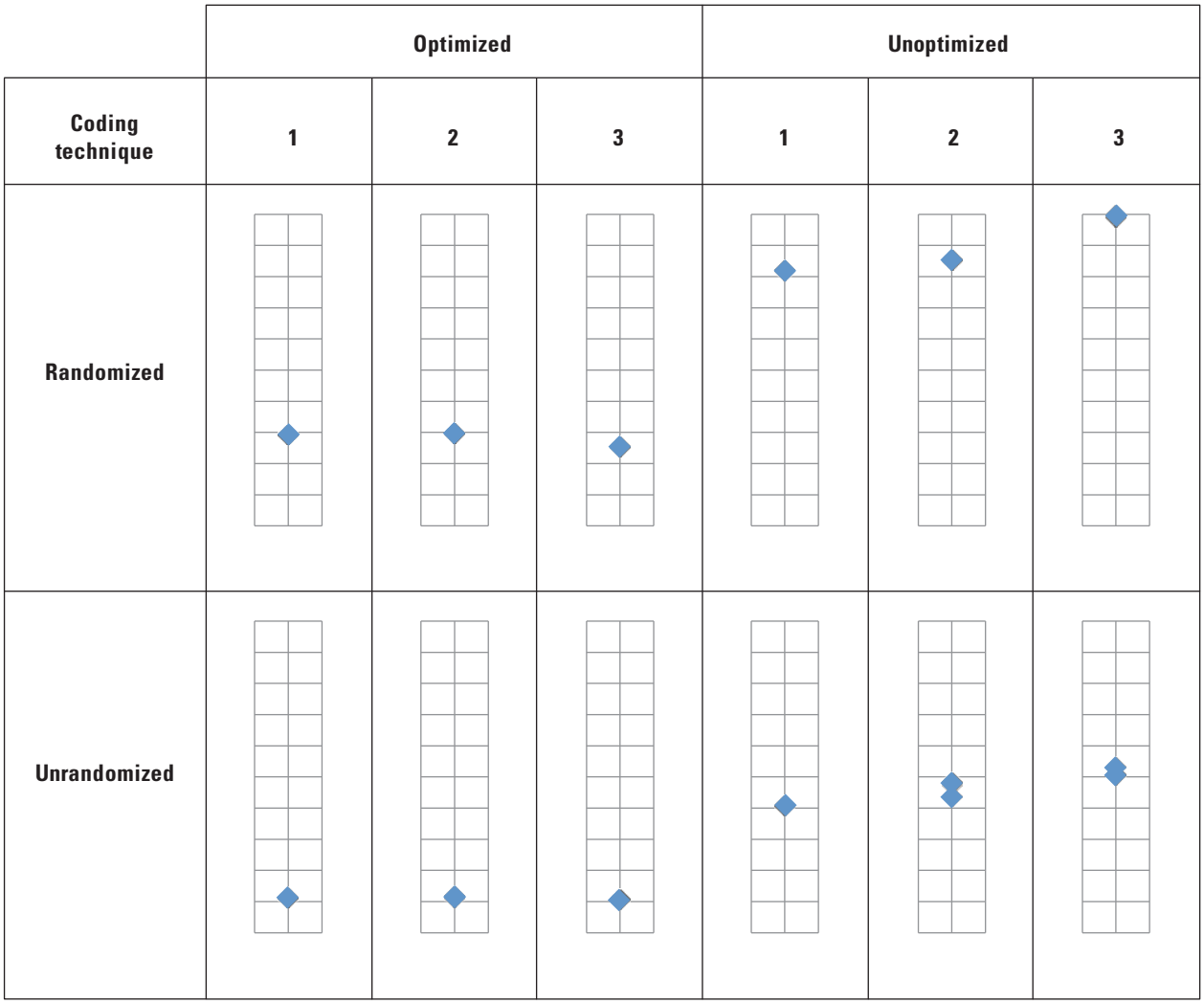

Figure 2–16 Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host C—64-bit Windows—C language.

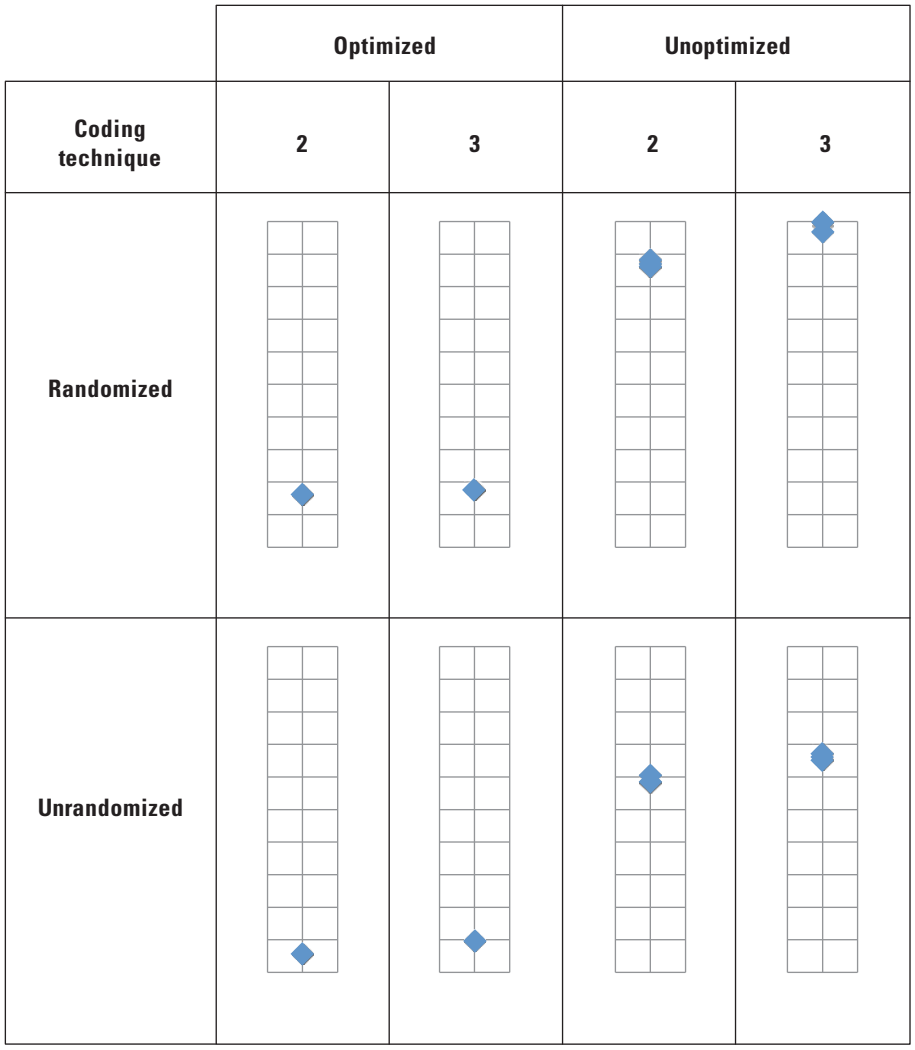

Figure 2–17. Boxplots of relative runtimes with coding techniques 2 and 3 on Host C—32-bit Windows—C++ language.

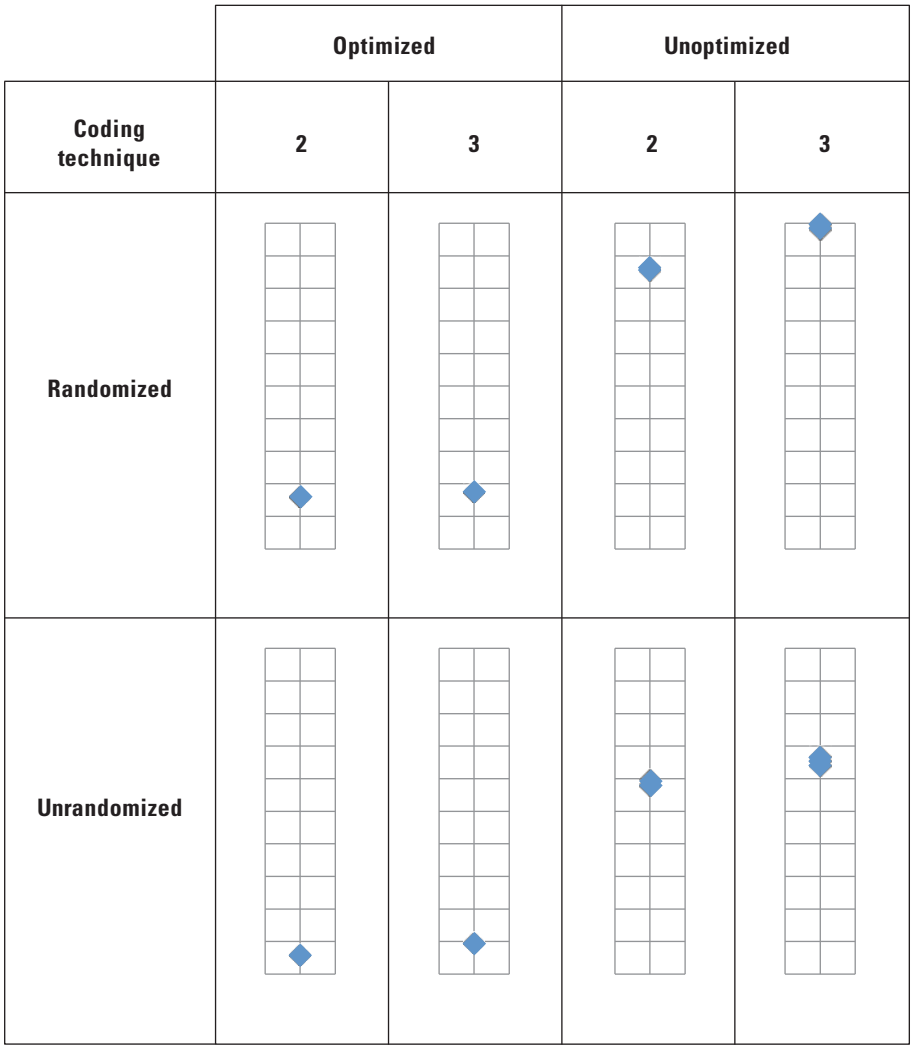

Figure 2–18. Boxplots of relative runtimes with coding techniques 2 and 3 on Host C—64-bit Windows—C++ language.

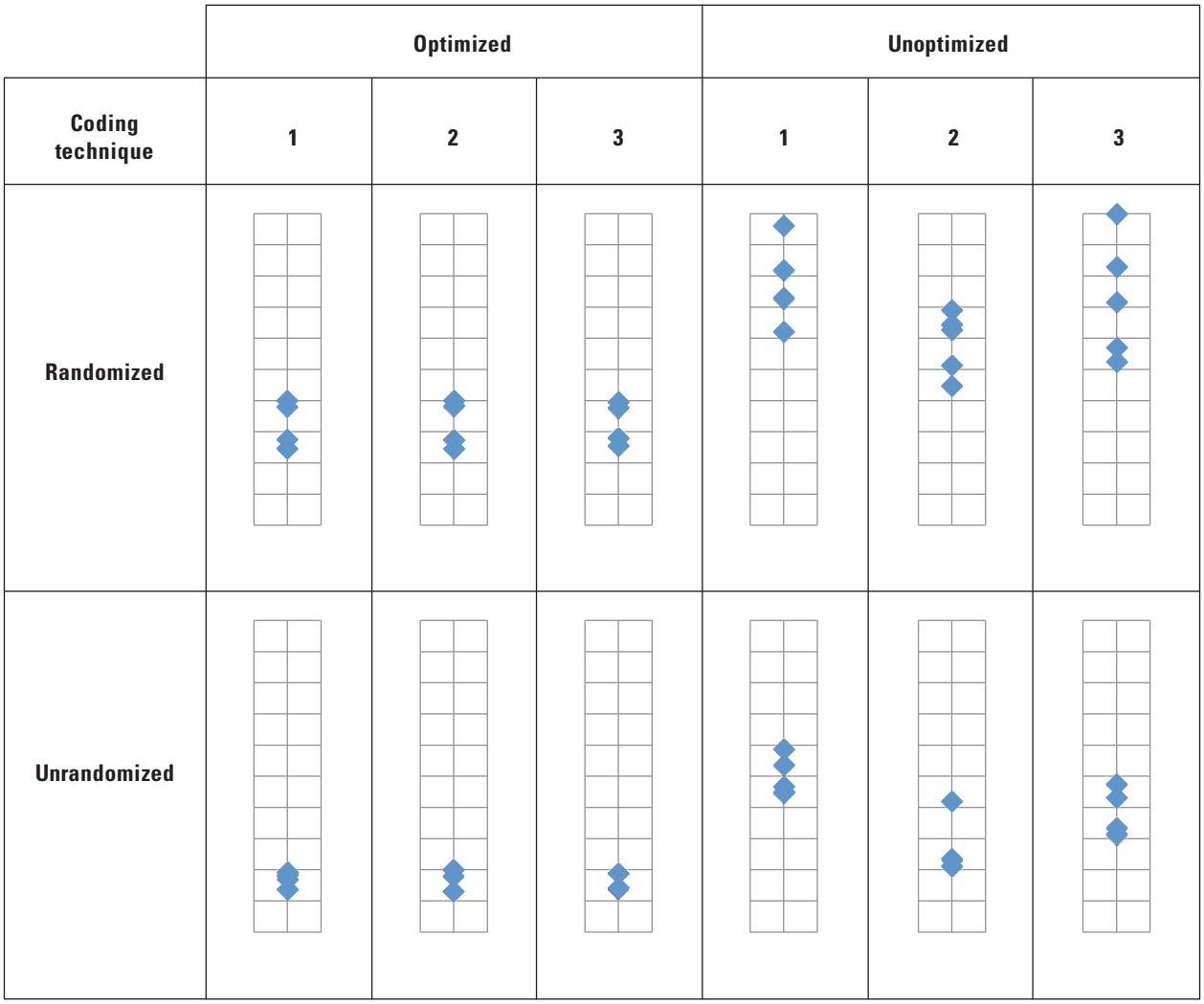

Figure 2–19. Boxplots of relative runtimes with coding techniques 1, 2, and 3 on Host C—64-bit Linux—C language.

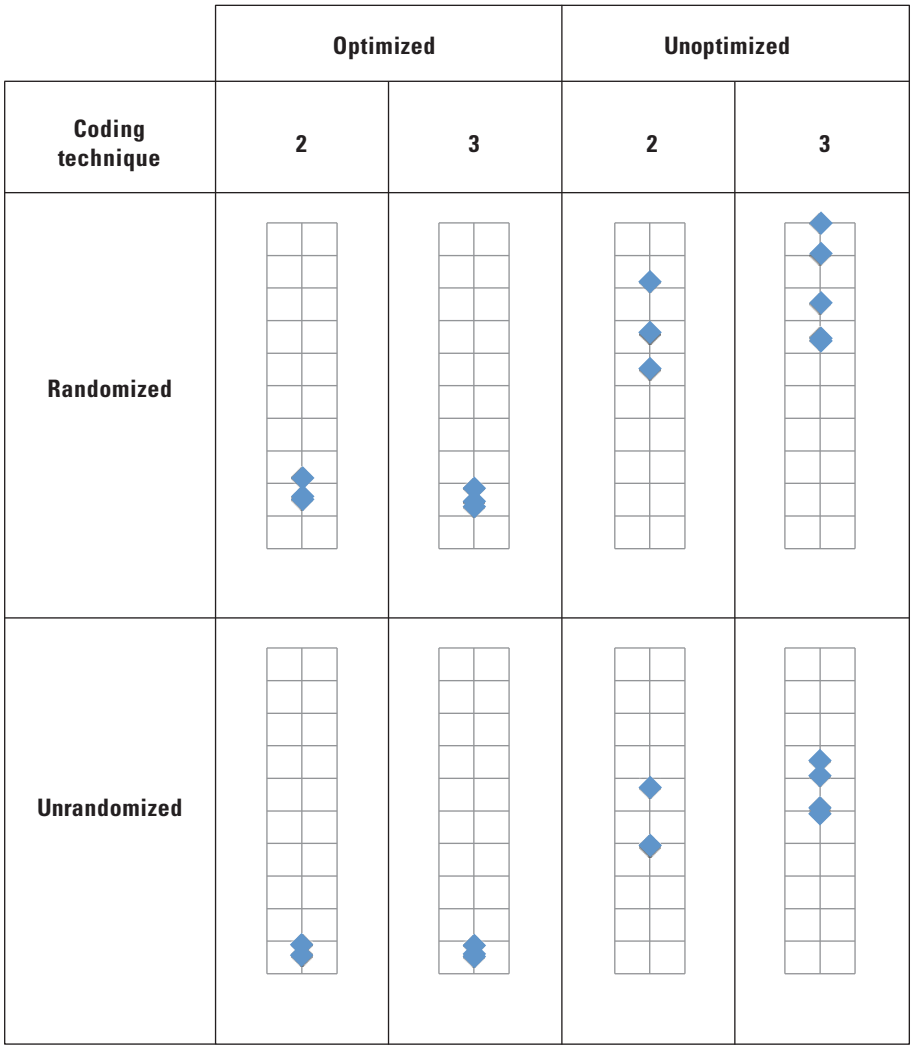

Figure 2–20. Boxplots of relative runtimes with coding techniques 2 and 3 on Host C—64-bit Linux—C++ language.

Appendix 3. Source Code for C Test Programs

Appendix 3 is provided as a compressed archive file named "ERC_Appendix_3.zip." This file contains the 28 files of source code in the C programming language used in the comparative study described and interpreted in this report.

Appendix 4. Source Code for C++ Test Programs

Appendix 4 is provided as a compressed archive file named "ERC_Appendix_4.zip." This file contains the 20 files of source code in the C++ programming language used in the comparative study described and interpreted in this report.

Appendix 5. Scripts and Code for Conducting Timing Tests on Linux

Appendix 5 is provided as a compressed archive file named "ERC_Appendix_5.zip." This file contains the scripts used under the Linux Bash shell to compile, run, and time the C and C++ test programs used in the comparative study described and interpreted in this report.

Appendix 6. Scripts and Code for Conducting Timing Tests on Windows

Appendix 6 is provided as a compressed archive file named "ERC_Appendix_6.zip." This file contains the batch and Powershell scripts run under Windows 7 to compile, run, and time the C and C++ test programs used in the comparative study described and interpreted in this report.

Manuscript approved December 19, 2016

For additional information regarding this publication, please contact:
Director, Eastern Geographic Science Center
U.S. Geological Survey
521 National Center
12201 Sunrise Valley Drive
Reston, VA 20192
Telephone: 703-648-4230
Or visit the Eastern Geographic Science Center website at http://egsc.usgs.gov/

Prepared by the USGS Science Publishing Network
Reston Publishing Service Center
Edited by Stokely J. Klasovsky
Illustrations by Caryl J. Wipperfurth
Layout by Jeffrey L. Corbett

ISSN 2328-7055 (online)
https://doi.org/10.3133/tm7E1