10.31341/jios.48.1.3 UDC 311.13: 004.912=93C++
Open Access Original Scientific Paper

Exploring the Access to the Static Array Elements via Indices and via Pointers — the Introductory C++ Case Expanded

Robert Logožar *robert.logozar@unin.hr*
Dpt. of Electrical Engineering
University North, Varaždin, Croatia

Matija Mikac *matija.mikac@unin.hr*
Dpt. of Electrical Engineering
University North, Varaždin, Croatia

Danijel Radošević *darados@foi.unizg.hr*
Faculty of Organization and Informatics
University of Zagreb, Varaždin, Croatia

Abstract

We revisit the old but formally still unresolved debate on the time efficiency of accessing the elements of 1D arrays via indices versus accessing them via pointers. To analyze that, we have programmed benchmarks of minimal complexity in the C++ language and inspected the machine code of their compilation in the x86 assembly language. Before exploring the performance, we briefly compared a few methods used for the execution time measurements. The results on the Wintel platform show no significant advantage in using pointers over indices except for some benchmarks and array (data) types. In other cases, the exact opposite may be true. The cause of this inconsistency lies in the compilation of the source code into the rather nonorthogonal x86 instruction set. Furthermore, the execution speed does not clearly relate to the instruction length. The parallel aim of this work is to provide a ground for further analysis and measurements of this kind using different compilers, languages, and computer platforms.

Keywords: static arrays, pointers, C/C++, accessing the array elements, benchmarks, execution time measurements.

1. Introduction

Algorithms and data structures are the very foundation of software. The simplest and most ubiquitous data structure, which resembles the organization of the computer's main memory itself, is the *array*. It is unavoidable in computer programming, and all general-purpose, high-level languages implement it one way or the other. The simplest way to access the array elements—which follows the mathematical notation of vectors and matrices—is by "subscripting" the arrays with their indices. In Pascal and the C language, this became possible also via pointers, which are the addresses of

the defined data types. The pointers allowed programmers to directly manipulate the memory addresses and their contents in a similar way possible in assembly languages but having left the organizational details to the compiler.

The fathers of the C language, B. Kernighan and D. Ritchie, in their C language "bible," known as K&R, devoted the whole of chapter 5 to the pointers and arrays and their relation [1]. In §5.3, they say: "*Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.*"

This claim must be echoing in the minds of many computer scientists and practical programmers who have read valuable classical literature and cared to write the most efficient code. Namely, if we follow its implicit suggestion, should we insist on accessing the array elements by using pointers? If so, what are the time efficiency improvements? Kernighan and Ritchie did not support their statement with any explanation or proof. They have even relativized it in the further elaboration on the array element access in the rest of that chapter and the textbook. Besides that, when trying to find out if someone else attempted to corroborate or dispute this thesis, we could not find any other systematic work or firm results covering this topic.

Having said that, this rather elementary dilemma may still intrigue the minds of many pedantic programmers who want to ensure that they write programming code that is as fast as possible. This may also intrigue the minds of the teachers who want to motivate their students to grasp the concept of pointers. In either case, to answer that properly, one should get a deeper insight into the topic and reach the final verdict only after the concrete time measurements. That is, for the appropriately tailored benchmarks of the two approaches, we must measure their *benchmark execution times*, which we will abbreviate as BETs.

We did some simple, preliminary time measurements roughly a decade ago. They showed that the access to the elements of one-dimensional arrays of some integer types (`short int`, `int`) via pointers was approximately 15% to 20% faster than the access via indices. We did not pursue the measurements more systematically nor did we investigate the causes of this behavior at that time. We left that — seemingly trivial research — for some "future work," which finally continues with this paper. Thus, our first aim is to provide an introduction for a detailed analysis of the provided benchmarks and their execution time measurements. At the same time, we will expose the first results obtained by using a modern integrated developing environment (IDE) on a relatively contemporary computer and today's common operating system (OS).

In this text, we expand our preliminary and abbreviated description of the topic given in the conference paper [2] and provide the full version of this exploration. Besides correcting several (minor) errata and noticed omissions, it includes a more detailed description of the results of compilation and the analysis of the obtained machine instructions, as well as a comprehensive insight into the measurement results.

Concerning the outline of this text, in section 2, we shortly expose the basics of the array data structure and the pointer data type and their implementation in the C/C++ languages. Section 3 presents and explains our benchmarks and exposes their assembly language code, often paying attention to several minute details to better understand the following performance measurements. Section 4 discusses the

methods, program setup, and prerequisites for the time measurements. There, we present and discuss the results of our BET measurements for assigning and retrieving array elements of several data types. Section 5 concludes this paper and opens several new directions and topics for future work.

2. Arrays and Pointers

Before elaborating on our benchmark details, we will briefly outline the basics of the arrays and pointers, emphasizing their implementation in C/C++.

2.1. Arrays

One-dimensional (1D) arrays serve for storing the series of n equal a_i elements, where the nonnegative integer index i spans through n consecutive values. For example, with $i = 1, 2, \dots, n$, the elements a_i could be interpreted as the components of vector $\boldsymbol{a}$ in an n-dimensional space. Following this mathematical notation, the standard syntax for accessing the array elements in high-level programming languages requires stating the array name and the element index. Based on that, the compilers ensure the run-time calculation of i-th array element memory address as:

$$A(i) = A_0 + l_T \times (i - i_0), \quad i = i_0,\ i_0 + 1, \dots, i_0 + n - 1\ . \tag{1}$$

Here, A_0 is the address of the starting element, the one with index i_0, and l_T is the size of the element data type (T) in the number of bytes (B)—here, of course, in their original meaning of (the size of) a basic memory location.

The value $x(i)$ of the i-th element is the memory $M_T\big(A(i)\big)$ content of the address $A(i)$ interpreted as the data type T,

$$x(i) = M_T\big(A(i)\big)\ . \tag{2}$$

2.2. C/C++ Arrays and Pointers

The C and C++ languages fix the starting index to $i_0 = 0$, so its value need not be subtracted from i. This leads to the simplest possible formula and the fastest possible address calculation:

$$A(i) = A_0 + l_T \times \mathrm{i}, \quad i = 0, 1, \dots, n - 1. \tag{3}$$

As hinted in the introduction (§1), the elements' addresses and contents can also be operated by the variable of the special, *pointer data type*, usually simply called *pointers*. They hold the addresses with assigned data types and can also be described as *typed addresses*.

Altogether, this leads to the three possible ways of accessing the array elements. We immediately write them in the syntax of the C/C++ languages [1] (1988), [3], [4], in the same way as they appear in the benchmarks' source code in Listing 1:

1) by using the element index: `iX[i]`;
2) by using the pointer arithmetic and then by dereferencing the obtained pointer: `*(pI0 + i)`;
3) by incrementing the pointer and dereferencing it: `*(++pI)`.

The first two ways are of the *random access* kind because they can access array elements directly via their indices, regardless of the previously accessed element. The third access applies only to the passage through successive array elements. To enable the comparison of the time efficiency between all three access types, here we restrict the consideration to only the successive passages through the array elements for all the access types. Of course, this kind of passage is also the most common in practice.

The above general deliberation is applicable to both *static* and *dynamic* arrays, but in the further text, we restrict our consideration to the former.

3. Our Benchmarks

In this section, we describe and analyze our benchmark routines. They are the core of our three C++ test programs created in MS Visual Studio, Community Version 2019 (further on, VS-CV 2019). The slight variations between the programs served to investigate the many peculiarities on which the execution times depend (§4.3). Because of the repetitive code for each data type, the programs built up to a bit more than 1000 lines each.

3.1. Benchmark Source Code

Listing 1 shows *our standard benchmarks* for the int type of 1D arrays. To avoid the user stack overflow, very large arrays—like ours—must be declared static or global. In each version of our test program, there are benchmark loops similar to those in Listing 1, but for the arrays of all the six standard data types — four integer and two floating-point types:

- `char` (1B), `short int` = `short` (2B), `int` (4B) (presented in Listing 1), and `long long int` = abbr. `llint` (8B);
- `float` (4B), and `double` (8B).

For each of the three described and implemented access methods, there are two benchmarks:

A. for storing a value into the `uI`-th array element, and
B. for retrieving (fetching) a value from the `uI`-th array element into a variable.

To investigate primarily the influence of different element-accessing mechanisms, we have kept the A and B types of operations as simple as possible. In A, instead of storing the quasi-random numbers into the array elements that serve as *l*-values, we assigned them the value of the unchanging `iRVal` variable. In action B, one and only one *l*-value, the variable `iLVal`, is assigned the values of the array elements.

```
 // Compiler Optimizations OFF!
// Definitions:
#define intMid     1111
#define intLrg 22222222
typedef unsigned int uint;
// Declaring the static int array:
const uint cuiN = 20000000;   // cuiN = 20 × 10^6.
uint uiN1 = cuiN;
static int iX[cuiN] = {0, }; // Array with 20 × 10^6 int elements.
iLVal = intMid, iRVal = intLrg;
// Pointers to the 0-th and last element:
int *pI0 = iX,  *pI1 = iX + uiN1;
```

```
// A. Storing into array elements
// 1) Via index, iX[uI]:
for (uint uI = 0; uI < uiN1; ++uI)
     iX[uI] = iRVal;
// 2) Via pointer arithmetic:
for (uint uI = 0; uI < uiN1; ++uI)
     *(pI0 + uI) = iRVal;
// 3) Via incrementing the pointer:
for (int* pI = iX; pI < pI1; ++pI)
     *pI = iRVal;
```

```
// B. Retrieving from array el.
// 1) Via index, iX[uI]:
for (uint uI = 0; uI < uiN1; ++uI)
    iLVal = iX[uI];
// 2) Via pointer arithmetic:
for (uint uI = 0; uI < uiN1; ++uI)
    iLVal = *(pI0 + uI);
// 3) Via incrementing the pointer:
for (int* pI = iX; pI < pI1; ++pI)
    iLVal = *pI;
```

Listing 1. Our standard benchmarks for accessing the elements of a (large) C/C++ `int` array by: 1) indices, 2) pointer arithmetic, 3) incrementing the pointer.

In the release version of the program, the compiler would optimize both actions if set so, especially B, because its outcome is the single assignment: `iLVal = iX[uiN1 - 1]`. That is why these benchmarks must run without any optimization.

The objection holds that such circumstances can be considered artificial and that the arrays could have been filled up in other ways. However, if our *r*- or *l*-values were more complex—to prevent the extreme optimization efficiency—the net effect of compilation without the optimization would have been the same. On the other hand, if the optimization were on, the investigation of the assembly language code (see the next subsection) would not be possible directly in VS-CV 2019, as it is now, for the non-optimized, debug version. Of course, some other versions of the benchmarks may be proposed, but not without complicating them at least a bit. For instance, in the assignment of the values to the array elements (action A), the *r*-value could be an element of another array filled with randomly generated numbers. However, in that case, both the *l*- and *r*-value would require accessing the array elements. Such a benchmark would perform double access to the array elements and thus equalize the actions A and B. While this is worth investigating, we leave it for future work and stick here to the proposed benchmarks with the described elementary actions.

Finally, a word about the use of the increment operator (++) that acts on an isolated *l*-value. We have been systematically using it in the prefix form, mostly for the sake

of uniformity and pedantry.[1] We will discuss other details of the concrete program implementation as needed.

3.2. Benchmark Machine Code Disassembled

When running the programs in the debug mode, the VS IDE enables the in-place presentation of the Intel assembly language instructions in symbolic form after disassembling the machine code of the debug version of the program [5]. The x86 (32-bit) and x64 (64-bit) versions of assembly languages are available, but in this paper, we focus on the still-standard 32-bit version. A concise review of the x86 assembly language can be found in [6], whereas the exhaustive reference is in [7]. For our six benchmark routines (A/B.1, 2, 3) with the array of `int` data type, this is shown in Listing 2.

3.2.1. Compilation of the `for`-loops (`int` arrays)

Of the six `for`-loops from our benchmarks in Listing 1, the first two of the A and B types (A/B.1 and A/B.2) have the standard `for`-loop *heads*, with (rising) indices. Because of the same source form, their compilation results in a series of the same eight (8) machine instructions, as presented for the A.1 `for`-loop head. There are only the expected slight differences in the instruction operands. The general-purpose registers used in A.1 are cyclically permuted in B.1 as follows: `EAX` → `ECX`, `ECX` → `EDX`, and `EDX` → `EAX`. Additionally, the loop compilations differ in the relative displacements of the jump addresses in their unconditional (`jmp`) and conditional (`jae`) jumps to the: i) loop *conditions* (`cmp` instruction), ii) *exits* (behind the last, `jmp` instruction in the loop body), and iii) *loop-expressions*, i.e., the index increments, starting at the second `mov` instruction.

To summarize, all these `for`-loops consist of eight (8) machine instructions in the loop head and one (1) additional at the end of the loop body, totaling nine (9) instructions, which are stored in the total o 43B (cf. type `int` benchmark A.1, the `for`-loop control part).

Concretely, in the A.1 `for`-loop head, the first `mov` places the initial, `0` value, stored in the instruction itself (immediate addressing mode) to the memory location of the loop index. The index address is expressed relative to the address contained in the base (frame) pointer register, `EBP`, which is constant during the execution of the main function. In the case shown here, `EBP = 212FFA38h`, so the address of the local `uI` index is: `A(uI) = EBP – 7E8h = 212FF250h`.

The next, `jmp` instruction, performs the jump to the condition-checking part of the loop head, which starts with the `mov` instruction at the address `main + 20E3h =`

[1] The discussion about the performance difference between the pre- and post-increment operators on the isolated *l*-values can be found e.g., in [8]. With the prefix `++`, our benchmarks mostly run from 0% to 2.5% faster. However, on rare occasions (for the longer data types!), they are sometimes slower, but within the standard deviation of the measurements (see sec. 4).

```
  // Listing 2
 // A. Storing into array elements
// A.1 Via index: iX[uI]
for (uint uI = 0; uI < uiN1; ++uI)
00C738B8  mov dword ptr [ebp-7E8h],0   ; uI = ML(EBP - 7E8h) ← 0.
00C738C2  jmp main+20E3h (0C738D3h)    ; GOTO 0C738D3h (loop condition).
00C738C4  mov eax,dword ptr [ebp-7E8h] ; EAX ← ML(EBP – 7E8h) = uI.
00C738CA  add eax,1                    ; EAX ← EAX + 1.
00C738CD  mov dword ptr [ebp-7E8h],eax ; uI ← EAX (uI ← uI + 1).
00C738D3  mov ecx,dword ptr [ebp-7E8h] ; ECX ← uI.
00C738D9  cmp ecx,dword ptr [ebp-77Ch] ; tmp ← ECX – ML(EBP – 77Ch) = uI – uiN1.
00C738DF  jae main+2106h (0C738F6h); If tmp ≥ 0 then GOTO 0C738F6h (exit loop).
        iX[uI] = iRVal;
00C738E1  mov edx,dword ptr [ebp-7E8h] ; EDX ← ML(EBP - 7E8h) = uI.
00C738E7  mov eax,dword ptr [ebp-738h] ; EAX ← ML(EBP - 738h) = iRVal (= intLrg).
00C738ED  mov dword ptr iX (0C826A8h)[edx*4],eax; iX[uI] = ML(iX + 4*uI) ← EAX
00C738F4  jmp main+20D4h (0C738C4h)      ; GOTO 0C738C4h (loop expression).
// A.2 Via pointer arithmetic: *(pI0 + uI)
for (uint uI = 0; uI < uiN1; ++uI) // As in A.1.
       *(pI0 + uI) = iRVal;
00C73B5B  mov eax,dword ptr [ebp-7F8h] ; EAX ← ML(EBP – 7F8h) = uI.
00C73B61  mov ecx,dword ptr [ebp-76Ch] ; ECX ← ML(EBP – 76Ch) = pI0 [= A(iX)].
00C73B67  mov edx,dword ptr [ebp-738h] ; EDX ← ML(EBP - 738h) = iRVal (= intLrg).
00C73B6D  mov dword ptr [ecx+eax*4],edx; *(pI0 + uI) = ML(ECX + 4*EAX) ← EDX
00C73B70  jmp main+234Eh (0C73B3Eh)    ; GOTO 0C73B3Eh (loop expression).
// A.3 Via incrementing the pointer: *(++pI)
for (int* pI = iX; pI < pI1; ++pI)
00C73DD1  mov dword ptr [ebp-808h], offset iX (0C826A8h); pI = ML(EBP - 808h) ← A(iX).
00C73DDB  jmp main+25FCh (0C73DECh)    ; GOTO 0C73DECh (loop condition).
00C73DDD  mov edx,dword ptr [ebp-808h] ; EDX ← ML(EBP – 808h) = pI.
00C73DE3  add edx,4                    ; EDX ← EDX + 4.
00C73DE6  mov dword ptr [ebp-808h],edx ; pI ← EDX (pI ← pI + 4).
00C73DEC  mov eax,dword ptr [ebp-808h] ; EAX ← pI.
00C73DF2  cmp eax,dword ptr [ebp-770h] ; tmp ← EAX – ML(EBP – 770h) = pI – pI1.
00C73DF8  jae main+261Ah (0C73E0Ah); If tmp ≥ 0 then GOTO 0C73E0Ah (exit loop).
       *pI = iRVal;
00C73DFA  mov ecx,dword ptr [ebp-808h] ; ECX ← ML(EBP - 808h) = pI
00C73E00  mov edx,dword ptr [ebp-738h] ; EDX ← ML(EBP - 738h) = iRVal (= intLrg).
00C73E06  mov dword ptr [ecx],edx      ; *pI = ML(ECX) ← EDX.
00C73E08  jmp main+25EDh (0C73DDDh)    ; GOTO 0C73DDDh (loop expression).

 // B. Retrieving from array elements
// B.1 Via index: iX[uI]
for (uint uI = 0; uI < uiN1; ++uI) // As in A.1.
        iLVal = iX[uI];
00C73A09  mov eax,dword ptr [ebp-7F0h] ; EAX ← ML(EBP – 7F0h) = uI.
00C73A0F  mov ecx,dword ptr iX (0C826A8h)[eax*4]; ECX ← ML(iX + 4*uI) = iX[uI].
00C73A16  mov dword ptr [ebp-784h],ecx ; iLVal = ML(EBP – 784h) ← ECX.
00C73A1C  jmp main+21FCh (0C739ECh)    ; GOTO 0C739ECh (loop expression).
// B.2 Via pointer arithmetic: *(pI0 + uI)
for (uint uI = 0; uI < uiN1; ++uI) // As in A.1.
        iLval = *(pI0 + uI);
00C73C9B  mov edx,dword ptr [ebp-800h] ; EDX ← ML(EBP – 800h) = uI.
00C73CA1  mov eax,dword ptr [ebp-76Ch] ; EAX ← ML(EBP – 76Ch) = pI0 [= A(iX)].
00C73CA7  mov ecx,dword ptr [eax+edx*4]; ECX ← ML(EAX+4*EDX) = *(pI0 + uI).
00C73CAA  mov dword ptr [ebp-784h],ecx ; iLVal = ML(EBP – 784h) ← ECX.
00C73CB0  jmp main+248Eh (0C73C7Eh)    ; GOTO 0C73B3Eh (loop expression).
```

```
 // Listing 2 continued.
// B.3 Via incrementing the pointer: *(++pI)
for (int* pI = iX; pI < pI1; ++pI) // As in A.3.
          iLval = *pI;
00C73F3D  mov ecx,dword ptr [ebp-810h] ; ECX ← ML(EBP – 7F0h) = pI.
00C73F43  mov edx,dword ptr [ecx]      ; EDX ← ML(ECX) = *pI.
00C73F45  mov dword ptr [ebp-784h],edx ; iLVal = ML(EBP – 784h) ← EDX.
00C73F4B  jmp main+2730h (0C73F20h)    ; GOTO 0C739ECh (loop expression).
```

Listing 2. Intel x86 assembly language code of the crucial parts of our benchmarks from Listing 1. The order of the code snippets is rearranged for the sake of comparison (see §4.2.1). In the code comments, $A(x)$ denotes the address of (variable) x and $M_T(A)$ denotes the contents on the address A interpreted in the data type T.

`00C738D3h`, where `main = 00C717F0h` represents the starting address of the C++ program main function.[2]

This `mov` prepares the operands for the `cmp` instruction behind it by moving the current `uI` index value to the `ECX` register.

Now `cmp` compares `uI` with `uiN1` variable value (placed at the address `EBP – 77Ch`,[3] by executing the operation `tmp ← uI – uiN1`, and setting the flags accordingly (cf. the x86 instruction reference in [6]). The next instruction, `jae`, performs the jump to the loop exit if the previous result is *above* or *equal to* zero. In other words, for `uI < uiN1`, there is no jump, and the loop enters its body. The body contains the statement `iX[uI] = iRVal;`, compiled into three instructions to be discussed soon.

The last instruction in A.1 `for`-loop is the `jmp` to the address `00C738F4h`, which is the unconditional jump to the loop-expression part of the loop head at the address `0C738C4h`. Here, the index `uI` will be incremented. Because in the x86 instruction set there are no explicit arithmetic operations with the operands in memory, the index value is first moved to the `EAX` register, and after that, it is incremented by the instruction `add eax, 1`.

The next (`mov`) instruction places the obtained result as the new value of index `uI`. After that, the execution of the `for`-loop comes again to its condition-checking part, starting with the preparatory `mov` instruction described at the top of this paragraph.

After observing this control part of the `for`-loop more closely, a reader might wonder why the second `mov` in the pair of the move instructions (the one at the address `00C738D3h`) is not omitted, i.e., why the `cmp` instructions were not of the form:

```
cmp dword ptr [ebp-7E8h], dword ptr [ebp-77Ch].
```

[2] From that address on, in our case at the addresses `00C717F0h – 00C71806h`, there are the first seven instructions of the `main` function, which implement the standard calling convention (see e.g., [6]). As the result of some compilations, the symbolic address of the next (`push`) instruction, `__$EncStackInitStart = main + 00C71807h`, can also appear in this place.

[3] In the VS 2019, the address of a local variable can be found by entering the C/C++ expression for getting its address into the address field of one of the four available disassembly memory windows. Here, `uiN1 = cuiN, A(uiN1) = &uiN1 = 212FF2BCh`.

However, that is invalid because the `cmp` does not allow the second operand to be accessed via a memory address! On the other hand, if it were a constant, which would happen if the `cuiN` (declared as `const`) were the upper limit of the index, as in the following condition,

```
for (uint uI = 0; uI < cuiN; ++uI),
```

then the compiler would indeed omit the "extra" `mov` instruction and prepare the `cmp` instruction as:

```
cmp dword ptr [ebp-7E8h], 1312D00h .
```

Here, the constant value of `cuiN = 1312D00h = 20 000 000d` is stored in the instruction itself. This case exemplifies how the x86 set of instructions is *not orthogonal*, i.e., how many operations are not allowed with an arbitrary addressing mode.

In our old benchmarks, we wrote the `for`-loops in just the above way, and their loop heads were translated into the following seven (7) instructions, with their lengths in bytes in parenthesis:

- `mov`(10), `jmp`(2), `mov`(6), `add`(3), `mov`(6), `cmp`(10), `jae`(2), plus
- `jmp`(2) at the end of the loop body,

adding up to eight (8) instructions in 41B.

This total is one (1) instruction and 2B less than in our standard benchmarks (cf. Table 1). However, whether such loops will run faster or not remains to be determined (§4.3.3). Nevertheless, they do not resemble the general case in which the upper limit can and often will be a non-constant value, so we have abandoned them.

A reader can also detect that the index value was incremented in `EAX`, moved back to the `uI` variable, to be moved into another (`ECX`) register. Why? Would it not be better to operate `cmp` with `EAX` as its first operand if the index value is already there? Furthermore, if we upgrade this by changing the first instruction to `mov EAX, 0`, then the "second" `mov` (second after `add`) is not needed at all. This simplification leads to code optimization, which is beyond the scope of this paper.

To get back to the present state of our benchmark loops, in Listing 2 we observe that the `for`-loop in A.3 benchmark is very similar to the previous ones and that the one in B.3 is equivalent. They both have the same number of instructions (8 + 1), with the same lengths as the loops in the first two benchmarks (cf. Table 1). Furthermore, all instructions in the A.3 `for`-loop are of the same type as those in A.1, with just a few minor differences. In the first instruction, the second operand must be addressed differently because it assigns the address of the `iX` array to be the initial value of the `pI` pointer: `pI ← A(iX) = iX = 00C826A8h`. The second instruction executes the jump to the condition-checking part of the loop head (at the address `00C73DECh`), which — as in A.1 — starts with the preparatory `mov` instruction. It moves the value of `pI` into `EAX`. Then `cmp` executes the subtraction `tmp ← pI – pI1`. `jae` checks the flags and if `tmp ≥ 0`, it exits the loop, else (`CF = 1`), the execution proceeds to the loop body. `jmp` at the end of the body directs the execution to the loop-expression, which first moves the value of `pI` to `EDX`, then increases `EDX` for 4, because `sizeof(int) = 4`, and finally stores the enlarged value into `pI`.

Overall, the differences between manipulating the pointer and the indices are only subtle. The compilation of all `for`-loops in our standard benchmarks gives the same number of instructions, with the same operations and the same or very similar addressing modes, resulting in the instructions of the same lengths. From this, one could expect them to execute within the nearly same time, but whether this is so — we still have to see.

3.2.2. Compilation of the array-element-accessing statements (`int` arrays)

In the `for`-loop bodies of all the presented benchmarks, there is just one C/C++ statement. First, we focus on the compilation of this statement in the three benchmarks of type A, in which an *r*-value provided by a variable `iRVal` is stored into the `uI`-th array element of the integer array `iX`.

In the benchmarks A.1 and A.3 for the `int` array, this is accomplished by three (3) and in the benchmark A.2 by four (4) `mov` instructions (cf. Listing 2 and Table 1). Therefore, we compare A.1 and A.3 cases first. In A.1, the three `mov` instructions together are 19B long, and in A.3 (only) 14B. The first two instructions in both benchmarks are not only of the same length but also of the exact same type. In A.1 (A.3) benchmark:

- the first `mov` instruction places the value of the index `uI` (pointer `pI`) into `EDX (ECX)`;
- the second `mov` instruction, in both A.1 and A.3, places the value of the local variable `iRVal = intLrg = 22222222d = 0153158Eh`, stored at the address `A(iRval) = EBP – 738h`, into `EAX (EDX)`.

The difference is only in the third `mov` instruction. In A.1, it moves the value of `EAX (= iRVal)` to the address of the `uI`-th array element, which is formed by the index addressing (eq. 3). In this case:

```
A(iX[uI]) = iX + 4*uI = 00C826A8h + 4*EDX.          (4)
```

It requires a 7B-`mov` instruction, which is 1B longer than the previous two. Namely, besides the information of the source operand (here `EAX`), it must store the information of the index register (`EDX`), the length of the (integer) array element (4), and—as the longest data—the 32-bit address of the `iX` array.

In A.3, the third `mov` instruction in the loop body is much simpler, and because of that, it is 5B shorter. Namely, the address of the `uI`-th array element is already prepared in the `pI` pointer (which was moved into `ECX` in the first `mov` instruction). So, the content of `EDX` moves to the address shown by `pI`.

To summarize this briefly, in the third `mov` instruction, we note an apparent simplification in benchmark A.3, comparing it to benchmark A.1. However, keeping in mind that this 32-bit code will be executed on the 64-bit platform (and particularly on the 64-bit processor!), it remains to see if the shorter and simpler last `mov` instruction will bring some speed benefits.

As for the loop body of the benchmark A.2, the four instructions are 6B, 6B, 6B, and 3B long, in total 21B, i.e., one instruction and 2B (7B) more than in A1 (A3). The first `mov` instruction places the value of `uI` into `EAX`. A careful reader will note that

Instruc. – op. mnem. (bytes)		*A:* `Arr. el. = const r-value`			*B:* `l-value = Afrr. el.`		
Array type	*for-loop*	A.1 `Arr[i]`	A.2 `*(p0+i)`	A.3 `*(++p)`	B.1 `Arr[i]`	B.2 `*(p0+i)`	B.3 `*(++p)`
char	ctrl:	Equiv. to int A.2.	Equiv. to int A.2.	Analog. to int A.3.	Equiv. to int B.1.	Equiv. to int B.2.	Sim./Analg. to int A.2.
	body:	mov(6), mov(6), mov(6): 18B. (Use of CL reg.)	mov(6), add(6), mov(6), mov(2): 20B.	mov(6), mov(6), mov(2): 14B. (Use of CL reg.)	mov(6), mov(6), mov(6): 18B. (Use of CL reg.)	mov(6), add(6), mov(2), mov(6): 20B.	mov(6), mov(2), mov(6): 14B. (Use of AL reg.)
short	ctrl:	Equiv. to int A.1.	Equiv. to int A.1.	Analog. to int A.3.	Equiv. to int B.1.	Equiv. to int B.2.	Analog. to int B.3.
	body:	mov(6), mov(7), mov(8): 21B. (Use of AX reg.)	mov(6), mov(6), mov(7), mov(4): 23B. (Use of CX.)	mov(6), mov(7), mov(3): 16B. (Use of DX reg.)	mov(6), mov(8), mov(7): 21B. (Use of CX reg.)	mov(6), mov(6), mov(4), mov(7): 23B. (Use of CX.)	mov(6), mov(3), mov(7): 16B. (Use of DX reg.)
int	ctrl:	mov(10), jmp(2), mov(6), add(3), mov(6), mov(6), cmp(6), jae(2); +jmp(2): 43B.	Equiv. to A.1, but with permuted reg.: EAX→ECX→EDX (↑ ← ← ↓)	Simil. to A.1, but with permuted reg.: ECX→EAX→EDX (↑ ← ← ↓)	Equiv. to A.2.	Equiv. to A.1.	Equiv. to A.3.
	body:	mov(6), mov(6), mov(7): 19B.	mov(6), mov(6), mov(6), mov(3): 21B.	mov(6), mov(6), mov(2): 14B.	mov(6), mov(7), mov(6): 19B.	mov(6), add(6), mov(3), mov(6): 21B.	mov(6), mov(2), mov(6); 14B.
ll-int	ctrl:	Equiv. to int A.1.	Equiv. to int A.2.	Analog. to int A.3.	Equiv. to int A.1.	Equiv. to int A.2.	Equiv. to int B.3.
	body:	mov(6), mov(6), mov(6), mov(7), mov(7): 32B.	mov(6), mov(6), mov(6), mov(6), mov(3), mov(4): 31B.	mov(6), mov(6), mov(6), mov(2), mov(3): 23B.	mov(6), mov(7), mov(7), mov(6), mov(6): 32B.	mov(6), mov(6), mov(3), mov(4), mov(6), mov(6): 31B.	mov(6), mov(2), mov(3), mov(6), mov(6): 23B.
float	ctrl:	Equiv. to int A.1.	Equiv. to int A.1.	Equiv. to int A.3.	Equiv. to int A.1.	Equiv. to int A.1.	Equiv. to int B.3.
	body:	mov(6), movss(8), movss(9): 23B.	mov(6), mov(6), movss(8), movss(5): 25B.	mov(6), movss(8), movss(4): 18B.	mov(6), movss(9), movss(8): 23B.	mov(6), mov(6), movss(5), movss(8): 25B.	mov(6), movss(4), movss(8): 18B.
double	ctrl:	Equiv. to int A.1.	Equiv. to int A.2.	Sim./Analg. to int A.1.	Equiv. to int A.1.	Equiv. to int B.1.	Analog. to int A.3.
	body:	mov(6), movsd(8), movsd(9): 23B.	mov(6), mov(6), movsd(8), movsd(5): 25B.	mov(6), movsd(8), movsd(4): 18B.	mov(6), movsd(9), movsd(8): 23B.	mov(6), mov(6), movsd(5), movsd(8): 25B.	mov(6), movsd(4), movsd(8): 18B.

Table 1. The x86 machine instructions of our six standard benchmarks and their lengths for the six standard array types. The benchmarks' `for`-loops are divided into the control part (the loop head and the unconditional jump at the end of the loop body) and the loop body contents. For the two parts, the instructions are represented as: `opMnem`(*l*), where `opMnem` = the operation mnemonic, *l* = the instruction length in bytes. Equiv. = equivalent up to the address specs., Analog./Sim. = as equiv. but adapted to the data type/with a different addressing mode.

this `uI` index is different from the one in A.1, because all indices are declared locally to their `for`-loops. Here, `A(uI) = EBP – 7F8h` [as stated in §3.2.1, we omitted the A.2 `for`-loop head compilation because it is equivalent to that in A.1, except for the circular replacement of the registers (cf. Table 1)].

The next `mov` places the starting `pI0 = A(iX)` pointer value into `ECX` [`A(pI0) = EBP – 76Ch`]. The third `mov` stores `iRVal` into `EDX`. Now everything is prepared for the action of the fourth `mov`, which will form the address of the `uI`-th element by using the index addressing in a very similar way as in eq. 4, but with all the operands in registers:

```
A(iX[uI]) = pI0 + 4*uI = 00C826A8h + 4*EAX.          (5)
```

To recapitulate this shortly: despite one more instruction in this case, the last instruction—which performs the final assignment of the value to the `uI`-th array element—is relatively short (3B) and probably very effective.

The three benchmarks of type B remain, in which the `uI`-th array element value is assigned to the *l*-value provided by the variable `iLVal`. Again, the assignment statements in B.1 and B.3 bodies consist of three (3), and in B.2 of four (4) `mov` instructions. Thus, we start with the former two benchmarks.

In the B.1 loop body, the three `mov` instructions are the same type as those in A.1 and have the same lengths, but the last two are in the reversed order and have different source and destination operands. The first `mov` places the value of `uI` (now with `A(uI) = EBP – 7F0h`) into `EAX`; the second stores the value of `iX[uI]` into `ECX`; the last one moves the value from `ECX` into the `iLVAl` variable; `A(iLVal) = EBP – 784h`.

The analogous situation is in B.3. Again, the instructions there are of the same type as those in A.3 but permuted. First, the pointer `pI` value is moved to `ECX` [`A(pI) = EBP – 810h`]. The second `mov` retrieves the contents from the address in `ECX`, i.e., the value of `*pI`, and stores it into `EDX`, making this crucial fetch of the array element very effective. Finally, the third `mov` places the `EDX` value into `iLVal`.

Similarly, in the B.2 loop body, the first two instructions do the same thing as those in A.2 but with `EDX` and `EAX` as destinations. The third, the shortest one (3B), stores the `*(pI0 + uI)` value into `ECX`. The fourth instruction moves this value from `ECX` to `iLVal`. Judging solely by the number of instructions and their lengths, this solution is longer, but—as suggested previously—its time efficiency may also depend on other factors.

3.2.3. Compilation of the benchmarks with arrays of other types

A detailed inspection of the machine instructions for all other array (data) types would be very voluminous. For them, we will only briefly comment on the data summarized in Table 1, in which the above-elaborated `int`-type array benchmarks serve as the referent ones.

Regarding the implementation of the control part of the `for`-loops, the reader can note that it is the same for all 36 benchmark variations. Because they have indices (counters) and incrementing pointers within the range of the `int` type, they are all compiled to the same instructions, with the same lengths, possibly differing from each other only as follows:

Equiv. — stands for the cases where the implementation is *equivalent* in every way except in the concrete choice of the general-purpose registers (GPRS) and the address offsets.

Analog. — stands for the *analogous* cases, which differ from the equivalent only in that some instruction operands are adapted to the concrete data type, e.g., by changing the address increment according to the data type length: `incr = sizeof(type)`.

Sim. (similar) — stands for the case with the same types of instructions but with different addressing modes.

The inspection of Table 1 confirms that all `for`-loop implementations are — if not equivalent — either analogous or similar in the above sense. The biggest difference, attributed as "similar," concretely means that only the first instruction in the `for`-loop head is different (see the comparison of the loop heads for A.1 and A.3 benchmarks in §3.2.1).

In the implementation of the single C/C++ statement within the loop body, there is much greater diversity, as we have already shown for the `int` arrays. Similarly, the observation that the type-2 benchmarks (A.2 and B.2) have one instruction more and the type-3 benchmarks (A.3 and B.3) have the shortest total length of their instructions also holds for the other array data types. Besides that general remark, we note that with the same number of instructions for the `short`- and `int`-type arrays, the former has a net length that is systematically 2B longer. These instructions use the 16-bit `X`-type registers (`AX`, `CX`, `DX`). The situation levels up for the `char` type. These benchmarks have instructions of the same or faintly shorter length than those for the `int` type, and they use the lower byte of the `X` registers (`AL`, `CL`).

The benchmarks with the `llint` arrays systematically have two (2) instructions more than the corresponding ones of `int` type and are, because of that, considerably longer. In the x86 mode, the VS-CV 2019 compiler produces the x86, 32-bit machine code without using the available 64-bit (`R`) registers, so moving this type of data requires the engagement of two 32-bit registers instead of one, and the use of the standard `dword mov` instructions (`double word` = 32-bit word). In our concrete example, the value of the `llint` variable `llRVal` is stored into the two registers: `ECX:EDX`, of which the left (right) holds the more (less) significant half of the 64-bit value.

The remaining two types of arrays are of the floating-point type. For accessing the array elements of the `float` (`double`) type, the compiler uses `movss` (`movsd`) instructions to move the 32(64)-bit contents to the lowest (lower) portion of the 128-bit `xxm` registers (in our case, it was the `xmm0` register (Intel 2022-2). From their short description in Table 1, we can see that the numbers of these instructions are equal to the numbers of instructions in the corresponding benchmarks for the `int` arrays and that their length is equal for both the 32-bit `float` and 64-bit `double float` type.

3.2.4. A Glimpse to the x64 Compilation

In VS-CV 2019, the default compilation option is for the x86 set of machine instructions, as a still de-facto standard for many sorts of applications. In addition, there is also the x64 compilation, which translates the C++ source code into the Intel's x64 set of 64-bit instructions. To explore this option thoroughly, we would have to inspect the additional instructions in the i64 set, which have access to the quadword (64-bit), R-registers. It would also require the analysis of the compiled machine code similar to the above one. We will leave this out of this work, and here just briefly describe the crucial differences between the 64-bit version and the 32-bit version of the assembly code.

The structure of the `for`-loops in our benchmarks remained the same: there are eight (8) instructions in the loop heads and one (1) unconditional jump at the end of the loop bodies. For our referent, `int` type, and the A.1 benchmark, the 3B-instruction `add eax,1` is now replaced with the 2B-instruction `inc eax,1` (making us wonder why it was not used in the x86 code, too). Furthermore, the loop indices are stored locally with the displacement counted now from the `RSP` register (the 64-bit stack pointer), instead of from the `EBP` register. Overall, the x64 version of our `for`-loops is realized similarly to before. They have the same number of instructions (9), and their total length is 42B, 1B less than in our standard, x86 version loop. This structure holds for all `for`-loops with indices. The `for`-loops with the incrementing pointers are organized somewhat differently. They have, in total, ten (10) instructions and a much longer length of 55B.

As for the assignment statement in the loop body, it is compiled into four (4) instructions: `mov`(7), `lea`(7), `mov`(4), `mov`(3), totaling 23B, and having one (1) instruction and 4B more than the x86 version. Here, all instructions but the first one work with the `R`-type of registers for preparing the operand addresses. For the benchmarks with the `int`-type arrays, their *r*-values and array elements are manipulated according to their type, i.e., as the 32-bit memory and register values. The same instructions, but ordered differently, are in the loop bodies of the B-type benchmarks.

The situation is very similar for the shorter (integer) types, for which the loop bodies have one instruction more than in the x86 version. Likewise, for the `float` type, the loop body of A.1 benchmark is realized by four (4) instructions: `mov`(7), `lea`(7), `movss`(6), `movss`(5), totaling 25B, and having one (1) instruction and 2B more than the x86 version. Generally, it is clear that for the types equal to or shorter than 32-bits, the use of x64 architecture does not improve the structure of the compiled machine code.

The benefits of the 64-bit architecture should—if anywhere—become obvious for the 64-bit data types. Indeed, the loop body of the `llint` A.1 benchmark is realized by (only) four (4) instructions: `mov`(7), `lea`(7), `mov`(8), `mov`(4), totaling 28B, which is one (1) instruction and 4B less than in the x86 machine code. However, for the type `double` of A.1 benchmark, the instructions in the loop body are: `mov`(7), `lea`(7), `movsd`(9), `movsd`(5), totaling 28B, which is one (1) instruction and 5B more than in the x86 version. The rest of this analysis might be presented elsewhere.

3.2.5. Possible Benchmark Pitfalls

The benchmarks written for this research were not intended to perform some standard operations[4] but to be as simple as possible and thus eliminate all unnecessary consumption of the processor time unrelated to the purpose of this testing. In such a reduction, one must pay attention to the generality of the written program code. Otherwise, it may easily happen that the obtained benchmarks favor some of the testing options before others. We will illustrate this in the following example.

In the early versions of our A-type benchmarks, instead of the `iRVal` variable, we have used a numerical constant as a source of the value to be stored in the array element. In a slightly simplified version, shown here in Listing 3, the constant is an integer defined by `intLrg`, as can be seen in the C/C++ assignment statement above the corresponding assembly language code.[5] Unlike the code in Listing 2, the `jmp` instruction on the bottom of the `for`-loop body is not shown because the `for`-loop context is omitted here. In all three cases, there is one `mov` instruction less than in our standard benchmarks (in Listing 2). The numbers and the lengths of the instructions (in parenthesis) are as follows:

A.1 – 2 instructions: `mov`(6), `mov`(11), in total 17B;

A.2 – 3 instructions: `mov`(6), `mov`(6), `mov`(7), in total 19B;

A.3 – 2 instructions: `mov`(6), `mov`(6), in total 12B.

```
 // WARNING! Not a general case because of assigning intLrg, which is de-
// fined as a numerical constant (see Listing 1) and is addressed literally.
// A.1 Storing into:
iX[uI] = intLrg;
00426078 mov eax,dword ptr [ebp-0A20h] ; EAX ← ML(EBP – 0A20h) = uI.
0042607E mov word ptr iX (03D6E850h)[eax*4], 153158Eh
                      ; iX[uI] = ML(iX + 4*uI) ← 153158Eh = 22 222 222d.
 //    ...           ...
// A.2 Storing into:
*(pI0 + uI) = intLrg;
004262E2 mov eax,dword ptr [ebp-0A7Ch]
004262E8 mov ecx,dword ptr [ebp-0A64h]
004262EE mov dword ptr [ecx+eax*4],153158Eh
 //    ...           ...
// A.3 Storing into:
*pI = intLrg;
00426571 mov eax,dword ptr [ebp-0AD8h] ; EAX ← ML(EBP – 0AD8h) = pI.
00426577 mov dword ptr [eax],153158Eh  ; *pI = ML(EAX) ← 153158Eh = 22 222 222d.
//     ...           ...
```

Listing 3. Intel x86 assembly language code of the B-type array element assignments with the less general *r*-value, in the form of numerical constant.

[4] Though the A-type benchmarks do perform the initialization of the array elements to the same value.

[5] In the test programs, we use two such constants: one for the `for`-loops pre-runs (§4.2.2), and another for their time-measuring runs.

The first `mov` instruction in A.1 and A.2 places the value of index `uI` into `EAX`. In A.3, the first `mov` places the value of the pointer `pI` into that register. The second `mov` in A.1 (A.3) finishes the job, by storing the value of the numerical constant (immediately addressed) `intLrg = 153158Eh = 22 222 222d` into the place of the `uI`-th (next) array element. In A.1, it is done by a more complex, 11B-`mov` instruction because of the longer form of the index addressing (4B are needed for the address of `iX`, and 4B for the stored literal). In A.3, the same action is achieved by a 6B-`mov` instruction, thanks to the fact that the fully formed array-element address was already in `EAX`. In A.2, the second of the three `mov` instructions must store the value of `pI0` into `ECX`, and after that, the third `mov` instruction stores the literal at the array element address formed by the index addressing. One might expect that, in this case, the A-type benchmarks would perform faster. We will comment on this in the next section (§4.3).

Another example of a pitfall in the benchmark code was already commented in the last part of §3.2.1. In it, a variable declared as `const` (`cuiN`) has enabled immediate addressing mode and thus — comparing it to the use of a general variable — decreased the number of instructions for one. Besides that, the variables should always be (re)declared in local blocks to assure similar addressing modes in all benchmarks.

In conclusion of this section, when writing benchmarks, one should pay great attention to their generality and follow all the rules of good programming practices. Additionally, before applying the newly created benchmarks, it is important to check their assembly language form.

4. Execution Time Measurements

This section will first explain the methods used for our BET (benchmark execution time) measurements and the prerequisites needed for consistent and precise results. Then, it presents those results and comments on them.

4.1. Time-Measurement Methods

There are several ways to measure the elapsed time of certain program parts in C++. In our preliminary and motivational measurements (mentioned in sec. 1), we have used the MS Windows `SYSTEMTIME struct`, available in VS IDE after including the `windows.h` header file (details in [9]). The way to apply it for the measurements of the benchmark execution times is shown in Listing 4. Because its smallest time division is a millisecond (ms), this method is useful when the measured times approach the order of one second (s). In that ad-hoc measurement, we explored the `short` and `int` arrays with `700 0000h = 117 440 512d` elements, i.e., five times larger than now. Thus, the execution times on the older and slower computers did approach `1s`, so such accuracy was acceptable. Recently, we repeated similar measurements on the arrays with roughly five times fewer elements because, in the same program, we were investigating six different types of arrays, most of them with much larger element types. The execution times of those measurements became too short for this

method. Sometimes, especially when the operation system (OS) was loaded more heavily, the function `GetSystemTime` failed to return a proper value and the calculated difference became negative.

More precise time measurements should get more accurate "time stamps" from the hardware timers, which rely directly on the processor's time cycles [10]. Such are the methods (ii) and (iii) in Listing 4. In (ii), the `HRTimer` class uses the member function `QueryPerformanceCounter` to do the job. We used this method for the time measurements in [11], [12]. By assuming that its accuracy should approach the execution time of the order of (several) instruction cycle(s), we appraised its precision to be around 10ns. However, the comparisons with the other methods used here put a question mark on this optimistic "guesstimate" (see below).

```
// Declarations and definitions as in Listing 1.
//    ...   ...
// Variables for the elapsed times in ms.
float fDltTmSYS, fDltTmHRT, fDltTmHRC;

// i) Time measurement by SYSTEMTIME (SYS)
#include "windows.h"

// _SYSTEMTIME structures and ptrs. to struc.:
_SYSTEMTIME sT1, sT2, *pST1 = &sT1, *pST2 = &sT2;
GetSystemTime(pST1);      // Stopwatch on.
for (uint uI = 0; uI < cN; uI++)
     iX[uI] = intLrg;
GetSystemTime(pST2);    // Stopwatch off.
fDltTmSYS = (float)(pST2->wSecond - pST1->wSecond)*1000 +
            (float)(pST2->wMilliseconds - pST1>wMilliseconds);

// ii) Time measurement by CHRTimer class (HRT)
#include "HRTimer.h" //High-Resolution Time.
CHRTimer hrTimer;   // CHRTimer object.
hrTimer.StartTimer(); // Stopwatch on.
for (uint uI = 0; uI < cN; uI++)
     iX[uI] = intLrg;
fDltTmHRT = (float)hrTimer.StopTimer()*1.e3f; // Stopwatch off.

// iii) Time measurement by high_resolution_clock class (HRC)
#include <chrono>  // XYZ
using namespace std;
using namespace chrono;
duration<float, milli> durTDlt;
auto t1 = high_resolution_clock::now();
auto t2 = high_resolution_clock::now();
t1 = high_resolution_clock::now();// Stopwatch on.
for (uint uI = 0; uI < cN; uI++)
     iX[uI] = intLrg;
t2 = high_resolution_clock::now();// Stopwatch off.
fDltTmHRC = (durTDlt = t2 - t1).count();
```

Listing 4. Examples of the three time-measurement methods in C++, applied to the A.1 benchmark for `int` array: i) `_SYSTEMTIME` structures, ii) `CHRTimer` class, and iii) the `high_resolution_clock` class.

The third time-measurement method tested in this work used the `high_resolution_clock` class from the C++ `std::chrono` library [13]. To get the elapsed time, the programmer uses the template class member `duration`, with which she defines a variable to accept the time difference (`t2 – t1`) of the recorded time points in desired time units. Since our results are of the order of tens of milliseconds, we have chosen that unit. There is a warning about the possibly inconsistent and inferior implementation of this class in different libraries and compilers, as mentioned in [13]. However, we have kept it anyway without investigating how it is realized in our case.

In a separate, short test C++ program, we have compared the results of the three time-measuring methods on the A-type benchmarks 1, 2, and 3 for the `short` and `int` types. On the "outmost side" of the benchmarks, we placed the "measuring points" for the `SYSTEMTIME`, in the middle, the points for the `HRTimer` class, and closest to the benchmarks, we put the `high_resolution_clock` class.

The first method, using `SYSTEMTIME`, gives only roughly correct individual results and nearly correct averages, which is not bad regarding its above-stated deficiencies. The second and third methods give very consistent results of satisfying accuracy for our measurements. However, it is quite surprising that they produce results with differences of the order of $10\mu s$, which is 10^3 times more than our estimate above and probably much more than it should be. The possible culprit is the dubious implementation of the method (iii). Furthermore, the absolute time values can especially have much greater errors than the above estimate ($10ns$) because of the overall imprecision of the clock crystal's frequency [10]. In these measurements, the relative time accuracy is the important one. It requires only the steadiness of the processor clock frequency and not its absolute precision. Anyhow, because the relative accuracy of method (iii) is more than satisfactory for our needs, we were using it as the method of choice for the time measurements in this investigation.

4.2. Time-Measurement Setup

To achieve consistent and accurate time measurements, a programmer should comply with some program- and system-wise conditions. Our approach tends to measure the "average best results," i.e., the optimal benchmark execution times for the given computer.

4.2.1. Benchmark Execution Order

In the test programs, we have placed the benchmarks for each of the six array types in an order different from the one shown in Listing 1. In that order, there are also the *pre-runs* (explained in the next section), as follows:

Repeat for the array element access type $k = 1, 2, 3$:
A. Pre-run A.k, then **A.k with BET** measurements;
B. Pre-run B.k, then **B.k with BET** measurements;
Repeat end.

In this way, we have simulated a somewhat more realistic situation in which the A- and B-type benchmarks exchange first, and then, there is a change in the type of array element access, according to the enumeration given in §2.2.

4.2.2. Pre-runs

The measured BETs can significantly depend on the momentary state of the memory system of today's computers with multitasking OSs. If the multilevel caches are already optimally filled with the data used in the benchmarks, the execution times will also be optimal, i.e., close to the shortest possible. If this condition is not fulfilled, the measured times can increase severely, making the results prone to erratic changes. The first way to deal with this problem is to exclude the "statistical outliers" from the measurements. In practice, it usually boils down to excluding one or a few highest and lowest values from the measurement set. Obviously, in our case, only the worst results are to be excluded.

Here, we have used another method — the benchmark *pre-runs*. They execute the benchmark fully or partly before measuring its execution time in the same or very similar way (cf. [11], [12]). Here, the pre-runs executed the benchmarks by assigning the array elements different number values. Obviously, because the longest BETs will mostly happen at the execution of the first benchmarks, these two approaches produce similar results.

In our C++ programs, the user can control the pre-runs by switching them on and off for the A- and B-types of benchmarks separately, which come one after the other for each array (data) type. The user can also change the number of iterations, i.e., she can specify the number of elements the pre-runs will access.

Without the pre-runs, we have observed that the A.1 benchmark — being the first of the six benchmarks in the row (§4.2.1) — in the first of the (10) measurements for each array type executes much longer, compared to the standard deviation of the rest of the measurements in the set. We have investigated this influence by running the release version of our Program 1 (directly from VS-CV 2019), first with both pre-runs on and then with both or just one of the pre-runs off. With both pre-runs off (F, F) and when only the B pre-run was on (F, T), the BETs of the first A.1 measurement in a series prolonged for approx.:

(F, F): `char`: 27%, `short`: 39%, `int`: 56%, `llint`: 179%, `float`: 62%, `double`: 116%;

(F, T): `char`: 13%, `short`: 29%, `int`: 50%, `llint`: 168%, `float`: 58%, `double`: 120%.

When A pre-run was on and B off, there were no delays. On the contrary, all the first measurements in the series were shortened by about 1.5%!?

If the pre-run executes only a part of the iterations, which the user can control by decreasing the number of iterations in them — and thus access only a part of our huge arrays — the slow-down effect is still present, more so the lesser the chosen number of iterations.

This brief analysis shows that the BETs without the pre-runs delay more for the arrays with the longer data types because they occupy larger memory space.

Moreover, the B-only pre-run (F, T) cannot cover for the delays of the upcoming A benchmarks of longer types. In addition, it turns out that only the pre-run before the first A.1 benchmark is necessary, while the rest of them are superfluous. However, in the present versions of our test programs, we have simply left all of them in the order specified in §4.2.1.

As a final remark on the pre-runs, although introducing them might seem like an artificial intervention into the programming code, without them, we would obtain unstable and less accurate results.

4.2.3. Computer and OS specifications

To complete the BET measurements, one must record the computer hardware and OS specifications. In this paper, we mainly present the measurements that are made on one computer with the following specifications:

Processor:	Intel® Core™ i7-8700K CPU @ 3.70 GHz, with 6 cores.
Installed RAM:	32.0GB
System type:	64-bit OS, x64-based processor.
Op. System:	Windows 10 Pro Education (build 19044.1766).

4.2.4. Measurement Preconditions

Our aim is to measure the execution times of our benchmarks only, as "pure" as possible. To achieve that, we want to eliminate all side effects that could interfere with the computer hardware and OS performance. For that matter, there are a few things to consider. The first is which program version to use — debug or release, and how to run it — from the IDE or by starting the executable code file.

The measurements showed that the influence of both of these options in our VS-CV 2019 is rather minor. By directly running the exe-files of the release and debug versions (reminding the reader that the compiling optimization was turned off) we have noticed no significant changes in the measured BETs. Typically, the time differences for most of the measurements were less than the standard deviation of the measurements. The debug versions for some of the six benchmarks for the six array data types were sometimes slower and sometimes even faster, resulting in the average delta-percentages of the measured corresponding mean values around 0.5% to 1.0%. However, the debug version is prone to delays for the arrays of certain data types, noticeably for the arrays of `char`, `short`, and not so often of the `double` types. These outbursts of delays have ranged chiefly from 10% to 13%, but sometimes escalating to even 20%, then raising the average delay (of all 36 mean values) for up to 1.5%.

The comparison of running the programs directly from the exe-files vs. starting them from the VS-VC19 IDE and leaving the IDE on during the test program executions gives similar results. The former approach does provide a bit shorter BETs

(0.3% to 1.0%), but its true advantage is better stability. Namely, the latter is prone to sporadic upsurges of BETs up to 10%.[6]

The conclusion is that these differences would be irrelevant were it not for rare but possible erratic delays in the inferior options, that is, in running the debug version and running either version from the VSCV-2019 IDE. Thus, as a precaution to achieve better precision, our choice for the final measurements was as follows:

- to run the release versions (without compiler optimizations);
- to start the exe-files with all other applications turned off.

Furthermore, for today's standard multitasking OSs, with the otherwise "standard" execution environment, we take the following precautionary procedure:

- turn off all user applications;
- disable Ethernet and WLAN;
- check the task manager for the possibly demanding background processes and try to turn them off;
- run the test program with benchmarks.

4.3. Results of the BET Measurements

With all the preconditions being fulfilled, we run our benchmark test programs. Once the user enters the required input parameters, the remaining free run of one program on our computer lasts approximately 25s. It performs a benchmark pre-run and a BET measurement, normally both through the arrays with 20×10^6 elements, it does that for six (6) different benchmarks, repeats the whole action ten (10) times to get the averages, and repeats the same thing for six (6) different data types of array. This totals to the $6 \times 10 \times 6 = 360$ pre-runs and the same number of measured iterations, resulting in 14.4×10^9 assignment operations on the array elements. This number is halved if both the A- and B-type benchmark pre-runs are turned off. Each program lists 6×10 BETs for the six array (data) types, with their averages and standard deviations.

4.3.1. BETs of Our Standard Benchmarks

The typical results of the BET measurements for our six standard benchmarks (from Listing 1), extracted from a single run of (test) Program 1, are presented in Table 2.

For the A- and B-type assignments and the three different methods of accessing the array elements, six average $\overline{\Delta t}_{BET}$ times and (corrected) standard deviations are calculated for the set of 10 non-successive BET measurements for each array type.[7] In the extra columns for the A/B.2 and A/B.3 benchmarks, which access the array

[6] Although we did not practice it, the VS-VC19 IDE can be turned off after launching the desired (debug or release) version of the test program, and before proceeding with the program, i.e., before entering the few required input values.

[7] With 20×10^6 iterations performed in ≈ 30ms, one loop iteration takes 1.5ns, or 5.55 clock cycles (CC) of our processor, executing $12 - 13$ machine instructions. This means that the working-core's pipeline has an average throughput of 2.25 instructions per CC.

elements by using pointers, we can track the relative difference of their BETs comparing to the A/B.1 BETs.

The measured times and relations between them are similar for the A- and B-type benchmarks marked with the same numbers. A/B.1 is only slightly slower than A/B.3, meaning that the successive access to the array elements by incrementing the pointers did not considerably shorten the execution, although the lengths of their loop bodies are shorter by roughly a quarter (cf. Table 1). The BETs of A/B.3 benchmarks are shorter than those of A/B.1, but only for the amounts comparable to the measurement standard deviations.

Quite a surprise is that the execution is fastest for the A/B.2 benchmarks (except for the `llint`), although the number of the instructions in their loop body is greater by one-third than those in A/B.1 and 3. In addition, their total length is roughly 10% (30%) longer than in A.3 (B.3). This example clearly shows how the number and the lengths of instructions do not directly dictate their execution times. The decrease is quite significant for the `short` and `int` integers and for both floating-point types, ranging from roughly 30% to more than 40%. The big exception for the `llint` type (+57%) is caused by the simultaneous lack of comparable improvement for its A/B.2 benchmarks and the sudden 38% (34%) decrease of its BETs of A.1 & 3 (B.1 &3), resulting in the significant raise of the relative values for A/B.2.

For the `char` type, the BETs of A/B.2 benchmarks are just a little faster than for A/B.1 & 3. Furthermore, they are $\approx 45\%$ longer than for all other, longer types, except the `llint`, which is quite surprising. Namely, one would expect the BETs for this data type to be similar to the (other) integers but not slower.

From the above deliberation and by looking at the absolute values of those times, we come to the following conclusion.

For our standard benchmarks (Listing 1), the fastest way to access the array elements of type:

- `char`, `short`, `int`, `float`, and `double` is by using **pointer arithmetic** [`*(p0 + i)`], as in A/B.2, though for `char` it is only slightly better;
- `llint` type is by using
  - **pointer incrementing** [`*(pI)`], as in A/B.3, or just a little bit slower by using
  - **indices** (`Arr[i]`), as in A/B.1.

4.3.2. BETs of the Modified Benchmarks

In our Program 2, we have modified the benchmarks from Listing 1 by replacing the assignment (=) operator with the compound addition-assignment (+=) operator. For those benchmarks, Table 3 contains the average values of their typical BET measurement set.

The times of the A- and B-type benchmarks are still close to each other, but not as much and as consistently as for the set of our standard benchmarks. In addition, there are a few greater discrepancies, as in the following cases: for `int` — cf. A.1 vs. B.1 and A.3 vs. B.3; for `char` — cf. A.2 vs. B.2. A big difference is also that now A.2

20×10^6 *elements*	*A: Arr. el. = r-Value,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms					*B: l-Value = Arr. el.,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms				
Array Type	A.1 `Arr[i]`	A.2 `*(p0+i)`	Δrel to A.1	A.3 `*(++p)`	Δrel *to* A.1	B.1 `Arr[i]`	B.2 `*(p0+i)`	Δrel *to* B.1	B.3 `*(++p)`	Δrel *to* B.1
`char`	35.50 ±0.45	33.68 ±0.43	−5.1%	34.90 ±0.47	−1.7%	36.96 ±0.32	35.36 ±0.37	−4.3%	35.95 ±0.29	−2.7%
`short`	35.45 ±0.45	23.89 ±0.43	−32.6%	35.05 ±0.48	−1.1%	36.78 ±0.36	22.63 ±0.26	−38.5%	35.90 ±0.26	−2.4%
`int`	38.45 ±0.50	22.65 ±0.34	−41.1%	37.81 ±0.34	−1.7%	36.78 ±0.53	23.28 ±0.23	−36.7%	35.96 ±0.32	−2.2%
`llint`	23.47 ±0.38	36.92 ±0.41	+57.3%	23.34 ±0.22	−0.6%	24.16 ±0.22	34.42 ±0.47	+42.4%	23.84 ±0.48	−1.4%
`float`	39.02 ±0.48	22.42 ±0.28	−42.5%	37.85 ±0.32	−3.0%	36.88 ±0.37	23.42 ±0.31	−36.5%	36.09 ±0.40	−2.1%
`double`	38.93 ±0.28	23.04 ±0.15	−40.8%	38.28 ±0.52	−1.7%	37.35 ±0.20	25.46 ±0.36	−31.8%	36.20 ±0.08	−3.1%

Table 2. Average $\overline{\Delta t}_{BET}$ (Benchmark Execution Times), for the benchmarks from Listing 1 (Program 1). The benchmarks run in the order stated in §4.2.1, in the release version of the program, on the computer specified in 4.2.3. $\overline{\Delta t}_{BET}$ and the corresponding standard deviations are calculated for 10 (nonconsecutive) measurements and expressed in ms (milliseconds). For the benchmarks A/B.2 and A/B.3, the second column gives the relative Δrel difference of their averages from the Δt_{BET} of the benchmark A/B.1. In the color print, those are marked in green (red) if $\leq -10\%$ ($\geq +10\%$).

20×10^6 *elements*	*A: Arr. el. = r-Value,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms					*B: l-Value = Arr. el.,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms				
Array Type	A.1 `Arr[i]`	A.2 `*(p0+i)`	Δrel to A.1	A.3 `*(++p)`	Δrel *to* A.1	B.1 `Arr[i]`	B.2 `*(p0+i)`	Δrel *to* B.1	B.3 `*(++p)`	Δrel *to* B.1
`char`	28.02 ±0.58	37.24 ±0.32	+32.9%	27.64 ±0.41	−1.4%	27.74 ±0.29	29.58 ±0.39	+6.7%	27.17 ±0.32	−2.1%
`short`	28.12 ±0.36	36.21 ±0.63	+28.8%	27.26 ±0.27	−3.1%	27.92 ±0.24	34.19 ±0.30	+22.5%	27.22 ±0.29	−2.5%
`int`	38.62 ±0.34	40.43 ±0.40	+4.7%	38.67 ±0.39	+0.1%	24.78 ±0.29	35.15 ±0.15	+41.8%	24.17 ±0.10	−2.5%
`llint`	35.21 ±0.38	36.01 ±0.40	+2.3%	35.02 ±0.17	−0.5%	30.28 ±0.29	35.00 ±0.35	+15.6%	29.44 ±0.23	−2.8%
`float`	39.07 ±0.66	40.95 ±0.61	+4.8%	38.49 ±0.65	−1.5%	40.53 ±0.35	39.55 ±0.74	−2.4%	39.36 ±0.37	−2.9%
`double`	39.61 ±0.82	41.55 ±0.86	+4.9%	39.14 ±0.80	−1.2%	40.83 ±0.79	40.16 ±1.08	−1.7%	39.61 ±0.58	−3.0%

Table 3. A set of the BET measurements for slightly altered benchmarks (Program 2): in the benchmarks from Listing 1, the assignment operator (`=`) is replaced by the compound addition-assignment operator (`+=`).

20×10^6 *elements*	*A: Arr. el. = r-Value,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms					*B: l-Value = Arr. el.,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms				
Array Type	A.1 `Arr[i]`	A.2 `*(p0+i)`	Δ*rel* to A.1	A.3 `*(++p)`	Δ*rel* *to* A.1	B.1 `Arr[i]`	B.2 `*(p0+i)`	Δ*rel* *to* B.1	B.3 `*(++p)`	Δ*rel* *to* B.1
`char`	22.29 ±0.35	34.71 ±0.56	+55.7%	21.94 ±0.24	−1.5%	36.76 ±0.36	35.45 ±0.31	−3.6%	36.04 ±0.31	−2.0%
`short`	22.40 ±0.22	39.48 ±0.42	+76.2%	22.91 ±0.33	−2.2%	36.60 ±0.29	22.35 ±0.33	−38.9%	36.16 ±0.57	−1.2%
`int`	22.47 ±0.20	38.26 ±0.43	+70.3%	22.05 ±0.29	−1.9%	36.63 ±0.33	23.50 ±0.39	−35.9%	36.02 ±0.43	−1.7%
`llint`	22.42 ±0.27	38.72 ±0.33	+72.2%	23.09 ±0.39	+3.0%	23.89 ±0.26	34.80 ±0.35	+45.7%	23.60 ±0.24	−1.2%
`float`	38.41 ±0.37	22.50 ±0.45	−41.4%	37.95 ±0.57	−1.2%	36.64 ±0.42	23.60 ±0.26	−35.6%	36.08 ±0.36	−1.5%
`double`	38.58 ±0.37	24.19 ±0.29	−37.3%	38.05 ±0.31	−1.4%	37.02 ±0.25	24.85 ±0.14	−32.9%	36.52 ±0.35	−1.3%

Table 4. A set of BETs for the benchmarks from our Program 3. These benchmarks differ from those in Listing 1 in that the A-type benchmarks store the numeric constants into the array elements (§3.2.4, Listing 3).

20×10^6 *elements*	*A: Arr. el. = r-Value,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms					*B: l-Value = Arr. el.,* $(\overline{\Delta t}_{BET} \pm s_{dev})$/ms				
Array Type	A.1 `Arr[i]`	A.2 `*(p0+i)`	Δ*rel* to A.1	A.3 `*(++p)`	Δ*rel* *to* A.1	B.1 `Arr[i]`	B.2 `*(p0+i)`	Δ*rel* *to* B.1	B.3 `*(++p)`	Δ*rel* *to* B.1
`char`	33.84 ±0.39	23.82 ±0.30	−29.6%	22.01 ±0.22	−35.0%	31.92 ±0.25	34.61 ±0.31	+8.4%	36.16 ±0.64	+13.3%
`short`	33.69 ±0.49	32.80 ±0.32	−2.6%	21.75 ±0.10	−35.4%	31.90 ±0.35	33.32 ±0.29	+4.4%	35.95 ±0.32	+12.7%
`int`	33.75 ±0.49	33.09 ±0.43	−2.0%	23.85 ±0.35	−29.3%	32.13 ±0.30	33.43 ±0.31	+4.0%	35.96 ±0.35	+11.9%
`llint`	33.01 ±0.47	31.70 ±0.46	−4.0%	23.26 ±0.28	−29.5%	30.65 ±0.42	30.89 ±0.32	+0.8%	23.70 ±0.30	−22.7%
`float`	33.31 ±0.47	37.66 ±0.27	+13.1%	37.93 ±0.31	+13.9%	31.83 ±0.29	33.40 ±0.29	+4.9%	36.16 ±0.39	+13.6%
`double`	32.54 ±0.46	35.73 ±0.48	+9.8%	38.08 ±0.36	+17.0%	32.46 ±0.48	34,16 ±0.40	+5.2%	36.63 ±0.40	+12.8%

Table 5. A set of BETs for our old benchmarks, now in Program 3-old. They differ from our standard benchmarks as those in Program 3, plus there is a numerical constant in the conditions of the `for`-loops with indices (A/B.1 & 2).

and B.2 are not the best access methods as they were for the standard benchmarks (except for `llint`). On the contrary, those are now either the worst or close to that, with `float`-B.2 being the only exception. As above, we summarize this as follows:

For our modified benchmarks (‘=’ → ‘+=’), the fastest way to access the array elements of

- all types except `int` is by **pointer incrementing** [`*(pI)`], as in A/B.3 (`int`-A.3 losing over `int`-A.1 for a mere 0.1%), or just a little bit worse by using
  - **indices** (`Arr[i]`), as in A/B.1.

An interesting observation is that these benchmarks — which not only assign the *r*-values but also add them to the *l*-values — execute faster than the assignment-only benchmarks in the following cases:

A/B.1 for `char`, `short`, and B.1 alone also for `int`, where the speed-up is in the range from roughly 20% to more than 30% (the best for `int`-B1);

A/B.3 for `char` and `short`, and B.3 alone also for `int`, with the speed-up from 20% to 25%.

On the other hand, we see that the execution times for the A.2 benchmarks lag from those of our standard benchmarks (in Table 2) for the first three integer array types, especially for the `char` and `short`, and then also for the floating-point types. B.2 is better for `char` but greatly lags for the same types as A.2.

In our third test program, the benchmarks are the same as in Listing 1, except that in the A-types, the elements are assigned numerical constants, as was discussed in §3.2.5. The aim was to investigate the influence of this less-than-general case. The BETs for these benchmarks are shown in Table 4. Since the B-type benchmarks are the same as in Program 1, one can check that their BETs follow those from Table 2 within the standard deviation values. As for the A-type benchmarks, the BETs of A.1 and A.3 are very close to each other and rather short for all integer types. In contrast, the times of the integer versions of A.2 are approx. 55% to 72% longer(!) than those of A.1 and A.3. For the floating-point types, the situation is reversed, with A.2 BETs being very short.

In comparison to the execution times of our standard benchmarks, these A.1 and A.3 BETs are shorter for more than 1/3 for the first three integers: for `char and short` 37%, for `int` 42%, while for `llint` and the floating-point types, these times are only slightly better (the most for `llint`, 4.5%). For the A.2 benchmark, the situation is reversed for the `short` and `int` arrays, which now have much longer BETs: `short` 65% and `int` 69%. For the other data types, the execution times are close to those of our standard benchmarks.

Summarily, the shorter loop body significantly improved the performance of the A.1 and A.3 benchmarks with the arrays of the first three integer types. For the A.2 benchmark, the opposite happens for the arrays of the first two integer types.

4.3.3. BETs of Our Older Benchmarks

To connect this analysis with our earlier investigation (cf. sec.), in Table 5, we present the results obtained by running the older version of our benchmarks, now marked as Program 3-old. In this program, there is another use of a constant: in the upper limit of the conditions in the `for`-loops with running indices (benchmarks A/B.1 & 2). We have discussed that in §3.2.1, showing that this kind of `for`-loop compiled into one instruction and 2B less than the `for`-loop of our standard benchmarks, hinting at the possible speed-ups. To recapitulate, in these A-type benchmarks, there are constants both in the array-element assignment statements and in the `for`-loop conditions, while

Arr. type	a) $\overline{\Delta t}_{BET,rel}$ *(Prog. 3-old / Prog. 1)*						b) $\overline{\Delta t}_{BET,rel}$ *(Prog. 1 mod. for-lp. hd/Prog. 1)*					
	A.1	A.2	A.3	B.1	B.2	B.3	A.1	A.2	A.3	B.1	B.2	B.3
`char`	−4.7%	−29.3%	−36.9%	−13.6%	−2.1%	+0.6%	−5.3%	+13.2%	−0.3%	−14.2%	−1.8%	+0.0%
`short`	−5.0%	+37.3%	−38.0%	−13.3%	+47.2%	+0.1%	−5.6%	+33.3%	−1.5%	−13.5%	+46.3%	+0.0%
`int`	−12.2%	+46.1%	−36.9%	−12.6%	+43.6%	0.0%	−14.1%	+65.2%	−0.1%	−13.6%	+43.9%	−0.2%
`llint`	+40.6%	−14.2%	−0.3%	+26.9%	−10.3%	−0.6%	+31.4%	−28.1%	−0.5%	+27.9%	−10.4%	−1.7%
`float`	−14.6%	+68.0%	−0.2%	−13.7%	+42.6%	0.2%	−15.8%	+65.1%	−0.3%	−14.1%	+41.5%	−0.3%
`dbl.`	−16.4%	+55.1%	−0.5%	−13.1%	+34.2%	+1.2%	−16.6%	+53.9%	−1.1%	−13.8%	+32.9%	+0.4%

Table 6. Relative BETs differences to the BETs of our standard benchmarks, from Table 2, for: a) our *old* benchmarks, now in Program 3-old; b) of the A/B.1 & 2 benchmarks as in Prog. 1, but with the constant in the `for`-loop conditions (`uI < cuiN`).

in B.1 & 2, only the latter applies. In fact, in the early version of the program, there were only A.1 and A.3 benchmarks for only the `short` and `int` types, but we have upgraded it to also include all the other benchmarks and array types as in the present test programs.

In the a) part of Table 6, we compare the BETs of these benchmarks to the corresponding execution times for our standard benchmarks. Despite the expectations to see only improvements — because of the diminution in the number and length of the instructions — we can see all three cases: decreases, invariabilities, and increases in the execution times. For instance, besides the usual unexpected behavior of the `llint` array elements in A(B).1 benchmarks, with the delay of 41% (27%), we also see large increases of the A/B.2 benchmark BETs for the first two integers and both floating-point types. On the other hand, there are quite significant improvements in A.3 for the first three integer types, etc.

Now, if we compare the "old" A.1 benchmark BETs for the integers (Table 5) to the corresponding ones from our Program 3 (Table 4), we may wonder why they are longer. Indeed, they have not only the assignments of constants in their `for`-loop bodies, as the benchmarks in Program 3, but they also have the constants in their `for`-loop heads. So why does their execution last longer for the integer types?

To solve this enigma, we have first modified the `for`-loop heads in our standard benchmarks (Program 1) of the A/B.1 and A/B.2 types to be the same as in this program. After checking the compilation in assembly language, we confirmed that all the modified `for`-loops now contain only seven (7) plus one (1) instructions, as explained in §3.2.1. Then, we measured the BETs of these modified benchmarks. Their relative differences from the BETs of our standard benchmarks are in the b) part of Table 6.

First, let us observe the A/B.3 benchmarks, whose `for`-loop heads were not changed in this modified program. We also detect that the corresponding BETs do not change. The changes do occur for the A/B.1 & 2 benchmarks, but again, not as we would expect, i.e., to be either close to zero or with negative values, because the only thing that happened in the machine code is its slight reduction. On the contrary,

besides the improvements, we also see much worse performance, e.g., for A/B.2 lower integers and both floating-point types, as well as for the `llint` type in A/B.1.

By comparing Table 6.b and Table 6.a, we can see that they are similar for A/B.1 & 2 benchmarks, except for the `char`-A.2 case, where there is a huge difference. However, we do not see that the slightly "simpler," old benchmarks are also faster. Quite the contrary, they are slower, though not for much. On the other hand, the tables differ significantly in A.3 for the first three integers, giving huge advantage to the old-style benchmarks, with ≈ 37% to 38% improvement vs. almost no improvement (≈ 0% to 1.5%) for the case presented in Table 6.b.

Thus, in the results of the old benchmark BETs, we do not see any improvements in all of them. Then, we see them in A/B.1 benchmarks (except for `llint`), but smaller than in their modified version in Table 6.b. That is, instead of the expected "synergy" of the shorter loop bodies and shorter `for`-loops, these times for integer types are now worse than for the benchmarks in Program 3 (Table 4)! All this happens although the compilation of the old benchmarks gives the `for`-loops with indices (in A/B.1 & 2) of the same form and size as those which made the improvements shown in Table 6.b and the loop bodies of the exact same form and size as the benchmarks in Program 3.

The resolution of this perplex is that the execution time depends not only on the machine instructions themselves but also considerably on the context they appear in. Thus, when comparing the old version of benchmarks A.1 to those in Program 3 (cf. Table 5 vs. Table 4), their execution times for the integer arrays do not decrease as one would expect — at least slightly. Instead of that, they increase by almost 50%. They are only somewhat better than the BETs of our standard benchmarks (Table 2). On the other hand, the BETs of the B.1 benchmarks do decrease by about 12% to 13%, but not so for the `llint`, etc.

In short, knowing the machine instructions and their lengths still does not give us the full information on how the machine code will be pushed through the processor's pipelines, how fast it will be decoded, and how efficiently it will be executed.

By focusing solely on the old benchmark's execution times (Table 5), we note that the A.3 benchmark BETs for *all* integer types—that is, now *including* the `llint` type, which has otherwise been a notorious exception—are from around 30% up to 35% shorter than for A.1s! As a kind of relief from all the previous complications, this result roughly confirms our earlier, much less systematic findings obtained on the same (old) benchmarks, but only on the arrays of the `short` and `int` types, as mentioned in sec. 1.

4.3.4. A Brief Overview of the BETs With x64 Compilation and on Other Computers

After the deliberation in §3.2.4, it may come as no surprise that the 64-bit, x64 machine code does not result in much faster executions. By inspecting Table 7, we see noticeable improvements from our standard benchmarks (Prog. 1) only for the A/B.1 and B.3 BETs, again, except for the `llint` array type. Here, this bad behavior of the 64-bit integer type is particularly odd. One would expect that the x64 compilation would improve at least the execution times of the compatible, 64-bit data

$\overline{\Delta t}_{BET,rel}$ *(Prog. 1 x64 / Prog. 1 x86)*						
Arr. typ.	A.1	A.2	A.3	B.1	B.2	B.3
`char`	−11.9%	+23.1%	+32.9%	−14.8%	−6.2%	−11.0%
`short`	−12.1%	+70.1%	+23.5%	−14.4%	+46.9%	−10.8%
`int`	−17.7%	+68.6%	+1.0%	−14.0%	+44.8%	−10.8%
`llint`	+35.9%	−2.3%	+53.0%	+34.9%	−0.2%	+37.4%
`float`	−18.2%	+72.4%	+2.9%	−14.8%	+44.8%	−11.4%
`double`	−18.2%	+62.2%	−5.4%	−14.3%	+33.4%	−9.2%

Table 7. Relative differences between the BETs of our standard benchmarks (Program 1) compiled to the x64 machine code and those compiled to the x86 code (shown in Table 2).

types, but it didn't happen. Additionally, the A.3 (B.3 for `llint`), and even more A/B.2 benchmarks, show much worse behavior than the corresponding x86 code.

In Program 2, the 64-bit compilation produces faster execution than the 32-bit compilation for all A-benchmarks but not for the first two integer types (not shown in a comparison table). The improvements range from around 10% up to 25%. The B-type benchmarks' BETs are from 10% shorter up to 20% longer. In Program 3, A.1 (B.1) benchmarks' BETs show slow-downs of about 40% (40% to 60%!) for all integer types and diverse behavior for the other benchmarks and array types.

For our old benchmarks (Program 3-old), the x64 machine code shows quite uniform BETs: for the A.1 benchmarks, they are in the approximate range from 31ms to 34ms, for A.2 from 32ms to 37ms, and for all the B-type benchmarks from 32ms to 34ms. Comparing them to the BETs of the corresponding x86 versions (Table 5), it turns out they are much faster (10%−12%) only for the B.3 benchmarks (again, except for the `llint` type!). Furthermore, since the BETs for the A.3 benchmarks are, in this case, similar to the other values, and in the x86 version, those were much better, their relative lags are quite large: for `char` 55%, `short` 53%, `int` 40%, and `llint` 38%. Because of that, in the x64 version of the benchmarks, the A.3-type of access is no longer significantly faster than A.1, as in their x86 version.

We have repeated the same measurements on a few other computers with the same Wintel platform. One of them has Intel® Core™ i3-6100U CPU @ 2.30GHz, with (only) two cores and relatively low energy consumption, and is running on the Windows 10 OS. It executes our benchmarks for roughly twice the time of the much more powerful computer described in §4.2.3. The BETs obtained on it follow the trends described in the previous text for all our test programs, with a few small discrepancies for certain benchmarks and array types. It also resembles the results of our old benchmarks, i.e., show the decrease of the BETs when using the pointer incrementing (A.3) over the use of the array indices (A.1), though slightly less than in our case: 25% for `char`, 32% for `short` and `int`, and 15% for `llint`.

The 64-bit compilations on this computer behave similarly to those on our primary computer, i.e., there are minor speed-ups in some cases but also slow-downs

in others. In addition, regarding the Program 3-old, there are no improvements in A.3 over A.1 benchmarks, same as on the primary computer.

5. Conclusion

After having struggled against the myriad of different cases, peculiarities, and surprises—most of which were without clear or, at least, simple rationale—we face the dilemma of what to say as a conclusion of this computing adventure! That is why it would not be a bad idea to recall why we started it in the first place. Namely, our initial intention seemed to be quite simple: to investigate whether one can improve the time efficiency of a program code by replacing the standard, index-based access to 1D-array elements with access via pointers. The mere volume of this paper shows that the answer to this question turned out to be neither unique nor simple and even less easy to explain.

Thus, we first recapitulate how we adopted our methods. We have built upon the ideas from our ad-hoc measurements from a decade ago, in which the array-element assignment statements were kept as simple as possible. In the initial benchmarks, there were only two types of accesses: via indices (No. 1), and via the incremented pointer (now No. 3). In their `for`-loop bodies, there was a single assignment statement with the array elements as the *l*-values. From the arrays of the `short` and `int` types only, these benchmarks were expanded to all six standard numerical array (data) types). Then, we also included the benchmarks with the mirror-symmetrical assignments, in which the array elements are the *r*-values. Thus, the two types of benchmarks were formed and named A and B, as shown in Listing 1. Both of them are simple, and because of that, their compilation with the optimization turned on can produce very simplified machine code, particularly for the B type. We have discussed this matter in detail in §3.1 and justified running and investigating the non-optimized machine code, especially if analyzing its disassembled version. These benchmarks can be regarded as the starting ones upon which the methodology will be built.

In the rest of section 3, we have explained the instructions of the compiled code and discussed a few pitfalls that we had encountered and had to deal with in this investigation. In doing that, we could notice the evident consequences of the non-orthogonality of Intel's x86 instruction set, in the sense that not all addressing modes are allowed with all operations. This non-orthogonality results in inconsistencies in how the source code can and will be compiled (§3.2.1, 3.2.2, 3.2.5).

However, after all the previous analysis, the attempt to understand and explain many varieties and peculiarities of the benchmark execution times (BETs) in section 4 was everything but an easy task. While we could clarify some of the obtained results by observing the number and length of instructions in the benchmarks' assembly code, some other results showed just the opposite from what would be predicted in that way (see §4.3 and, specifically, §4.3.3). A blatant example of such inconsistency is that shorter instructions do not necessarily perform faster. Knowing the complexity of the instruction pipelines in modern processors, especially those of the CISC types, this is far from a surprise, and it makes a plausible explanation for such deviant behavior even more elusive. It is not only that Intel's x86 instructions are of variable

length—which complicates the analysis of their execution times—but we have shown that these times significantly depend on the broader context within which these instructions appear, such as the data type of the array elements and the use of special operations and the corresponding (special) registers.

As we are approaching the end of this paper, can we conclude at least something? Is there some practical advice regarding the best method of accessing the array elements on a Wintel platform? By looking at the results of this research, we see that there is no simple conclusion and no unique best choice. The best method largely depends on the type of assignment operation in the loop body and the array (data) type. For example, we have managed to repeat the results of the early measurements on our old benchmarks and showed an even greater advantage of the use of the incrementing pointers over indices than before. However, the analysis showed that these results were obtained on the rather non-general benchmarks: they had constants in the crucial parts of their loops. Furthermore, the advantage occurs for only the first three integer types (Table 5, Table 6.a).

For our standard benchmarks, with the assignment-only statements in the loop bodies, we have given concrete advice in the bulleted list in §4.3.1 (based on the BETs from Table 2). For the modified benchmarks, with addition and assignment, the conclusion is summarized in the list in §4.3.2 (based on the BETs from Table 3).

Interpreting these results the other way around, we can say that using the indices can be as good as anything else if we primarily do iterative summations and if the improvement for a few percentages achieved by incrementing the pointer is irrelevant. Alternatively, we could say that the access via indices is considerably outperformed by the use of the pointer arithmetic in the case of the assignment-only operations, but not forgetting that the `llint` type is a persistent exception, etc.

In difference to that, one conclusion is rather simple: the x64 code does not improve the benchmark performance considerably, not even for the 64-bit types. Just the opposite! Table 7 shows that the execution times for most cases are worse than for the x86 code.

From all this, we can derive a conclusion of the presented subject on the standard Wintel platform: the best approach—especially in critical applications—is to measure the execution times of the intensively used piece of code as we did and to choose the access method with the best performance.

This conclusion, especially the last piece of advice, gave us a few hints on possible future work. First, the additional benchmarks should be written in a way that could not be trivialized by the compiler optimizations, even though this would prevent easy disassembly within the VS-CV 2019 IDE. Some of them could perform the standard array and matrix operations. Second, it would be interesting to make these measurements on some other platform(s), though—as things stand now—there are not many general-purpose processor brands to pick from! Namely, in the long-solitary "processor arena," with only one genuine player, there are no serious competitors at all. Some bright prospects that this could change are announced in [14]. Let us hope that the new architectural solutions, if possible, with an appropriate and more orthogonal instruction set, will contribute to the better performance consistency of our future computations.

References

[1] B. W. Kernighan, and D. M. Ritchie D. M, *The C Programming Language* 1st and 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1978 and 1988.

[2] R. Logožar, M. Mikac, and D. Radošević, "Exploring the Access to the Static Array Elements via Indices and via Pointers — the Introductory C++ Case," Proceedings of the Central European Conference on Information and Intelligent Systems, 33rd CECIIS, Dubrovnik, 2022, pp. 507–517.

[3] B. Stroustrup, *The C++ Programming Language*, 3rd ed. Murray Hill, New Jersey: AT&T Labs. 1997.

[4] H. Schildt, *C++: The Complete Reference*, 4th ed. New York: McGraw-Hill/Osborne, 2003.

[5] Microsoft; "View disassembly code in the Visual Studio debugger," *Microsoft, Documentation*, 2022. [Online]. Available at: https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-the-disassembly-window?view=vs-2022 [Accessed: Dec. 1, 2023].

[6] D. Evans, "x86 Assembly Guide." D. Evans, University of Virginia Computer Science, Spring 2006. [Online]. Available at: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html [Accessed: Dec. 1, 2023].

[7] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," *Intel, Developers,* 2022. [Online]. Available at: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html [Accessed: Dec. 1, 2023].

[8] Stack Overflow, "Is there a performance difference between `i++` and `++i` in C++," *Stack Overflow Questions*, 2009–2019. [Online]. Available at: https://stackoverflow.com/questions/24901/ [Accessed: Dec. 1, 2023].

[9] Microsoft, "SYSTEM-TIME structure (miniwinbase.h)," *Microsoft, Documentation*, 2021. [Online]. Available at: https://learn.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-systemtime [Accessed: Dec. 1, 2023].

[10] Microsoft, "Acquiring high-resolution time stamps," *Microsoft, Documentation*, 2022. [Online]. Available at: https://learn.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps [Accessed: Dec. 1, 2023].

[11] R. Logozar, "Recursive and Nonrecursive Traversal Algorithms for Dynamically Created Binary Trees," *Computer Technology and Application (David Publishing)*, vol. 3, no. 5, May., pp. 374–382, 2012.

[12] R. Logozar, "Algorithms and Data Structures for the Modeling of Dynamical Systems by Means of Stochastic Finite Automata," *Technical Gazette (Tehnički vjesnik)*, vol. 19, no. 2, Apr. –Jun., pp. 227–242, 2012.

[13] C++ reference, "std::chrono library, high_ resolution_clock class," *C++ reference, Date and time utilities."* [Online]. Available at: https://en.cppreference.com/w/cpp/chrono/high_resolution _clock [Accessed: Dec. 1, 2023].

[14] Apple, "Apple unveils M2, taking the breakthrough performance and capabilities of M1 even further," Apple, Newsroom, Press release, Jun., 2022. [Online]. Available at: https://www.apple.com/newsroom/2022/06/apple-unveils-m2-with-breakthrough-performance-and-capabilities/ [Accessed: Dec. 1, 2023].