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

Robert Logožar, Matija Mikac
University North, Croatia
Dpt. of Electrical Engineering
104. brigade 3, 42 000 Varaždin, Croatia
{robert.logozar, matija.mikac}@unin.hr

Danijel Radošević
University of Zagreb
Faculty of Organization and Informatics
Pavlinska 2, 42 000 Varaždin, Croatia
darados@foi.unizg.hr

***Abstract**. We revisit the old but formally still undecided 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 32-bit compilation in the x86 assembly language. Before the performance study, we have briefly compared a few methods used for the execution time measurements. There is no advantage in the use of pointers over indices except for some benchmarks and array (data) types, while for the others, the exact opposite may be true. The parallel aim was to provide a ground for the possible further analysis and measurements of this kind on different computers and platforms, and different languages.*

Keywords. Static arrays, pointers, C/C++, accessing the array elements, time measurement and efficiency.

1 Introduction

The array is the simplest and ubiquitous data structure that resembles the organization of the computer main memory itself. As such, 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 follow the mathematical notation of vectors and matrices — is by "subscripting" the arrays with their *indices*. In Pascal and the C language, that became possible also via *pointers*, which are the addresses of the defined data types.

The fathers of the C language, B. Kernighan and D. Ritchie, in their C language "bible," devoted the whole chapter 5 to the pointers and arrays and their relation (Kernighan & Ritchie, 1978, 1988, known as K&R). 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 the valuable classical literature and care to write the most efficient code. Namely, if we follow the implicit suggestion, should we insist on accessing the array elements by using pointers, and 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 see if someone else tried to corroborate or dispute this thesis, we could not find any systematic work or firm results on this topic from other authors.

To investigate this subject properly, one should get a deeper insight into it 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 shall 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 roughly 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 shall expose our first results obtained by using a modern integrated developing environment (IDE) on a relatively contemporary computer and today's common operating system (OS), but leaving the many details from this shorter version of our report.

Concerning the outline of this paper, in the next, 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. Section 4 discusses the methods, program setup and prerequisites for the time measurements. There, we present the results of our BET measurements and discuss them. 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 shall briefly outline the basics of the arrays and pointers, with an emphasis on 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:

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

Here, A_0 is the address of the starting element, the one with the 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 basic memory location.

2.2 C/C++ Arrays and Pointers

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

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

The value of the i-th element is the contents of the $A(i)$ address expressed as a certain data type.

As hinted in the introduction (§1), the element addresses and their contents can be operated also by using the variables of the *pointer data type*, or *pointers*. They hold the addresses with assigned data types and can be shortly described as *typed addresses*.

Altogether, this leads to the three possible ways of accessing the array elements, which we immediately write in the C/C++ languages (K&R 1988, Stroustrup 1997), 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 dereferencing the obtained pointer: `*(pI0 + i)`;
3) by incrementing the pointer: `*(++pI)`.

The first two are of the *random access* kind because their elements can be accessed directly via its index, 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 of all three access types, here we restrict the consideration only to such passage through the arrays, which is quite common in practice.

The above general deliberation is applicable to both *static* and *dynamic* arrays, but in the further text, we restrict our deliberation 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 them 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. 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) (in Listing 1), and `long long int` = abbr. `llint` (8B);
- `float` (4B), and `double` (8B).

To avoid the user stack overflow, the very large arrays, like ours, must be declared as `static` or as global.

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.

```
// Compiler Optimizations OFF!
#define intMid     1111
#define intLrg 22222222
typedef unsigned int uint;
// Defining the (static) int array:
const uint cuiN = 20000000;
uint uiN1 = cuiN;
static int iX[cuiN] = {0, };
iLVal = intMid, iRVal = intLrg;
// Pointers to the 0-th and last el.:
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: *(pI0 + uI).
for (uint uI = 0; uI < uiN1; ++uI)
    *(pI0 + uI) = iRVal;
// 3) Via icrementing the pointer: *(++pI).
for (int* pI = iX; pI < pI1; ++pI)
    *pI = iRVal;
// B. Retrieving from array elements
// 1) Via index: iX[uI]:
for (uint uI = 0; uI < uiN1; ++uI)
     iLVal = iX[uI];
// 2) Via pointer arithmetic: *(pI0 + uI).
for (uint uI = 0; uI < uiN1; ++uI)
    iLVal = *(pI0 + uI);
// 3) Via icrementing the pointer: *(++pI).
for (int* pI = iX; pI < pI1; ++pI)
    iLVal = *pI;
```

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

A. for storing a value in 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 observed 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 the action B, one and only one *l*-value, the variable `iLVal`, is assigned the values of the array elements.

In the release version of the program, the compiler could and would optimize both actions if set so, especially B, because its final 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 very artificial and that the arrays could have been filled up in some other ways. However, if our *r*- or *l*-values were more complex with the aim to prevent the extreme optimization efficiency, the net effect would have been the same as now. Furthermore, the investigation of the assembly language code (in the next subsection), would not be possible in VS-CV 2019, as it is for the non-optimized, debug version. Therefore, in this introductory investigation, we stick to the proposed benchmarks, with elementary actions.

We shall 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 integrated developing environment (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 (Intel 2022-1). 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 reader more interested in this topic can find a concise review of the x86 assembly language in (Evans 2022), whereas the exhaustive reference is in (Intel 2022-2). 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 eight machine instructions, which are *equivalent* to the A.1 `for`-loop head. I.e., they differ from each other only in the possible use of different alternative registers: `ECX` instead of `EAX`, `EDX` instead of `ECX`, and `EAX` instead of `ECX`. Additionally, (`jae`) jumps to the: i) loop *conditions* (`cmp` instruction),

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 (see §4.2.1).

```
 // A. Storing into array elements
// 1) Via index: iX[uI]
    for (uint uI = 0; uI < uiN1; ++uI)
00C738B8  mov   dword ptr [ebp-7E8h],0
00C738C2  jmp   main+20E3h (0C738D3h)
00C738C4  mov   eax,dword ptr [ebp-7E8h]
00C738CA  add   eax,1
00C738CD  mov   dword ptr [ebp-7E8h],eax
00C738D3  mov   ecx,dword ptr [ebp-7E8h]
00C738D9  cmp   ecx,dword ptr [ebp-77Ch]
00C738DF  jae main+2106h (0C738F6h)
          iX[uI] = iRVal;
00C738E1  mov   edx,dword ptr [ebp-7E8h]
00C738E7  mov   eax,dword ptr [ebp-738h]
00C738ED  mov dword ptr iX (0C826A8h)[edx*4],eax
00C738F4  jmp   main+20D4h (0C738C4h)
// 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]
00C73B61  mov   ecx,dword ptr [ebp-76Ch]
00C73B67  mov   edx,dword ptr [ebp-738h]
00C73B6D  mov   dword ptr [ecx+eax*4],edx
00C73B70  jmp   main+234Eh (0C73B3Eh)
// 3) Via incrementing the pointer: *(++pI)
    for (int* pI = iX; pI < pI1; ++pI)
00C73DD1  mov   dword ptr [ebp-808h], offset iX
                                     (0C826A8h)
00C73DDB  jmp   main+25FCh (0C73DECh)
00C73DDD  mov   edx,dword ptr [ebp-808h]
00C73DE3  add   edx,4
00C73DE6  mov   dword ptr [ebp-808h],edx
00C73DEC  mov   eax,dword ptr [ebp-808h]
00C73DF2  cmp   eax,dword ptr [ebp-770h]
00C73DF8  jae   main+261Ah (0C73E0Ah)
         *pI = iRVal;
00C73DFA  mov   ecx,dword ptr [ebp-808h]
00C73E00  mov   edx,dword ptr [ebp-738h]
00C73E06  mov   dword ptr [ecx],edx
00C73E08  jmp   main+25EDh (0C73DDDh)
 // B. Retrieving from array elements
// 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]
00C73A0F  mov ecx,dword ptr iX (0C826A8h)[eax*4]
00C73A16  mov   dword ptr [ebp-784h],ecx
00C73A1C  jmp   main+21FCh (0C739ECh)
// 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]
00C73CA1  mov   eax,dword ptr [ebp-76Ch]
00C73CA7  mov   ecx,dword ptr [eax+edx*4]
00C73CAA  mov   dword ptr [ebp-784h],ecx
00C73CB0  jmp   main+248Eh (0C73C7Eh)
// 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]
00C73F43  mov   edx,dword ptr [ecx]
00C73F45  mov   dword ptr [ebp-784h],edx
00C73F4B  jmp   main+2730h (0C73F20h)
```

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.

Summarily, 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 altogether 43B (cf. type `int` benchmark A.1, the `for`-loop control part).

We leave the in-detail explanation and analysis of the disassembled machine code for the extended version of this report. Here we just pay attention to a slight alteration of the `for`-loop conditions that results in different compilation results and execution times.

If the index upper limit in the `for`-loop conditions is a (variable declared as) constant, as in the following `for`-loop head:

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

then the compiler could and would omit the second `mov` instruction after `add` (at the address `00C738D3h`) and prepare the `cmp` instruction as

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

Here, the constant value of `cuiN = 1312D00h = 20 000 000h` is stored in the instruction itself. This is just an example of 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,
  - totaling eight (8) instructions in 41B.

This is one (1) instruction and 2B less than in our standard benchmarks. However, whether such loops will run faster or not, is still 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.

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 the one in B.3 is equivalent. They have the same number of instructions (8 + 1), with the same lengths as the loops in the first two benchmarks. 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.

Overall, the differences in manipulating the pointer from manipulating the indices are only subtle. The compilation of all `for`-loops in our standard benchmarks results in the same number of instructions, with the same operations, and the same or very similar addressing modes, resulting in their same lengths. A thorough analysis confirmed — what could have been expected in the first place — that this holds for the loops of the benchmarks with other five array (data) types as well. From this conclusion, 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 a *right*-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. 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 = 22 222 222d = 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. 1b). In this case:

`A(iX[uI]) = iX + 4*uI = 00C826A8h + 4*EDX`. (2)

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 5B shorter. Namely, the address of the `uI`-th array element is already prepared in the `iP` pointer (which was moved into `ECX` in the first `mov` instruction). So, the value of `EDX` is simply moved to the address shown by `iP`.

To summarize briefly, in that third `mov` instruction, we note a clear simplification in the benchmark A.3, comparing it to the benchmark A.1. However, having 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).

In B.1, the three `mov` instructions are of the same type as those in A.1 and have the same lengths, but they are placed in a different order and with 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`, and 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 B.2, the four mov instructions are of the same type as those in A.2. The first two of them do the same thing as those in A.2. The third, the shortest one (3B), stores the value of *(pI0 + iRVal) into ECX, and the last one 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 depend on other factors, as well.

As for the benchmarks with the other array (data) types, here we just summarize our findings after the analysis of all six (6) benchmarks for every of the six (6) standard data types. The implementation of the single C/C++ statement within the loop body is more diverse than of the loop control, as we have already shown for the int arrays. 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 holds also for the other array data types. Besides that general remark, while the number of instructions for the short- and int-type arrays is the same, the former has the 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 the 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 have systematically 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 that moving this type of data required the engagement of two 32-bit registers instead of one, and the use of the standard dword (double word = 32-bit word) mov instructions. 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. To access 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). The assembly code analysis showed 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.

3.2.3 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 machine instructions.

In this brief overview, we just summarize that the structure of the for-loops in our benchmarks remained the same after the x64 compilation: there are eight (8) instructions in the loop heads and one (1) unconditional jump at the end of the loop bodies. That is, there are nine (9) instructions with the total length of 42B, 1B less than in our standard, x86 version loop. This holds for all for-loops with indices. The for-loops with the incrementing pointers are organized somewhat differently, so that 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 to 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 but the first instruction work with the R-type of registers for preparing the operand addresses. As for the assignment statements for the int-type benchmarks, their *r*-values and the array elements are manipulated in accordance to their type, that is, 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, 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. Really, 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.

3.2.4 Importance of the Benchmark Analysis

The benchmarks written for this purpose were not intended to perform some standard operations (though the A-type benchmarks do perform the initialization of the array elements to the same value), but to be as simple as possible and thus eliminate all unnecessary consumption of the processor time that is not connected with the purpose of this testing. In such 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 the others.

For instance, in the early versions of our A-type benchmarks, instead of the `iRVal` variable, we have used a numerical constant as a value to be stored in the array element. This enabled the use of the immediate addressing mode for that operand, which resulted in one `mov` instruction less than in our standard benchmarks (in Listing 2). The numbers and the lengths of those instructions (in parenthesis) are as follows:

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

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

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

Another example of a pitfall in the benchmark code was already commented in §3.2.1.

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

4 Execution Time Measurements

In this section, we shall briefly present the methods used for our BET (benchmark execution time) measurements and the prerequisites needed to achieve consistent and precise results. Then we present these results and comment 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 MS 2022-1, example of application in Listing 3). With its smallest time division being a millisecond (ms), this method is useful when the measured times approach the order of one second (s). This was the case in our early measurements, where we measured the passage through the `short` and `int` arrays with `700 0000h = 117 440 512d` elements, i.e., five times larger than now, and on slower computers. In our present programs, this method is deserted.

More precise time measurements should get more accurate "time stamps" from the hardware timers, which rely directly on the processor's time cycles (MS 2012-2). Such are the methods (ii) and (iii) in Listing 3. In (ii), the `HRTimer` class uses the member function `QueryPerformanceCounter` to do the job. We used this method for the time measurements in (Logozar 2012-1/ 2012-2), overestimating its precision to 10ns. The third time measurement method tested in this work used the `high_resolution_clock` class from the C++ `std::chrono` library (C++ reference, 2022).

In a special, 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. Method (i) gives only roughly correct

Listing 3. 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.

```
// Declarations and definitions as in Listing 1.
//    ...       ...
// Variables for the elapsed times in ms.
float fDltTmSYS, fDltTmHRT, fDltTmHRC;
// i) Time mesurement 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 mesur. 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) 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();// On.
for (uint uI = 0; uI < cN; uI++)
     iX[uI] = intLrg;
t2 = high_resolution_clock::now();// Off.
fDltTmHRC = (durTDlt = t2 - t1).count();
```

individual results and nearly correct averages, which is not bad regarding its above-stated deficiencies. Methods (ii) and (iii) both give very consistent results of satisfying accuracy for our measurements. Surprisingly, though, their results differ for the order of magnitude of 10μs, which is much more than it should be if they are related to the processor's time cycles (C++ reference, 2022). Anyhow, because the accuracy of this method is more than satisfactory for our case, we have used it for the measurements in this paper.

4.2 Time Measurement Setup

To achieve consistent and accurate time measurements, a programmer should comply with some program- and system-wise conditions. In our approach, we tend 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.$\boldsymbol{k}$ with BET** measurements;
B. Pre-run B.k, then **B.$\boldsymbol{k}$ with BET** measurements;
End repeat.

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. To deal with this problem, here we have used the benchmark *pre-runs*. They execute the benchmark fully or partly before measuring its execution time, in the same or very similar way (Logozar 2012-1 and 2). 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.

4.2.3 Computer and OS specifications

To make the BET measurement results complete, it is essential to record the specifications of the computer hardware and OS. In this paper, we mostly 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.
OS:	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 — debug or release, and in which way — from the IDE or by starting the executable code file — to run. The measurements showed that the influence of both of these options in our VS-CV 2019 is rather minor. 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 give a bit shorter BETs (0.3% to 1.0%) but its true advantage is greater stability. Namely, the latter approach is prone to sporadic upsurges of BETs up to 10%.[1]

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, does that for six (6) different benchmarks, repeat the whole action ten (10) times to get the averages and repeat the same thing for six (6) different data types of array. This totals to the $6 \times 10 \times 6 = 360$ pre-run and the same number of measured iterations, resulting in 14.4×10^9 assignment operations on the array elements. If both the A- and B-type benchmark pre-runs are turned off, this number is halved. 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 in Table 1.

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.[2] In the extra columns for the A/B.2 and A/B.3 benchmarks, which

[1] 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.

[2] With 20×10^6 iterations performed in $\approx 30\text{ms}$, 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.

access the array 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 significantly shorten the execution, although the lengths of their loop bodies are shorter by roughly a quarter. The BETs for A/B.3 benchmarks are shorter than those for 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 of instructions and their lengths 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 the comparable improvement for its A/B.2 benchmarks and the sudden 38% (34%) decrease of its BETs for A.1 & 3 (B.1 &3), resulting in the great raise of the relative values for A/B.2.

For the `char` type, the BETs for 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 above 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 these benchmarks, Table 2 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 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 worst or close to that, with `float`-B.2 being the only exception. As above, we resume 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 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 1) for the first three integer array types, especially so for the `char` and `short`, and then also for the floating-point types. B.2 is better for char, but then greatly lags for the same types as A.2.

In our third test program, the benchmarks were the same as in Listing 1, except that in the A-types the array elements are assigned numerical constants, as it was discussed §3.2.4. The aim was to investigate the influence of this less-then-general case. The table with BETs for these benchmarks are not shown here, but will be just briefly discussed. Since the B-type benchmarks are the same as in Program 1, one can check that their BETs follow those from Table 1 within the standard deviation values. As for the A-type benchmarks, the BETs for A.1 and A.3 are very close to each other and rather short for all integer types. In opposite to that, the times of the integer versions of A.2 are approx. 55% to 72% longer(!) than those of A.1 and A.3, and slightly less so than of the A.2 types of our standard benchmarks. For the floating-point types, the situation is reversed, with A.2 BETs being very short.

Summarily, the shorter loop body greatly 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

In the older version of our benchmarks, now marked as Program 3-old, 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 resume, in these A-type benchmarks, there are constants in both the

Table 1. Average $\overline{\Delta t}_{BET}$ (Benchmark Execution Times), for the benchmarks from Listing 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 benchmark A/B.1 Δt_{BET}. 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`	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. A set of the BET measurements for slightly altered benchmarks (Prog. 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)`	*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%

array-element assignment statements and in the `for`-loop conditions, while 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 include also all the other benchmarks and array types as in the present test programs.

In Table 3, we compare the BETs of these benchmarks to the corresponding execution times for our standard benchmarks. Despite the expectations to see only improvements, we can see all three cases: decreases, invariabilities, and increases in the execution times, which do not necessarily follow the decrease in the number and length of the machine instructions. 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 the great decreases of BETs of A/B.2 benchmarks for the first two integers and both floating-point types. On the other hand, there are also quite significant improvements in A.3 for the first three integer types, etc.

By observing solely the old benchmark's execution times, it can be deduced 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! This result roughly confirms our early, much less systematic findings, obtained on the same (old) benchmarks, but only on the arrays of the `short` and `int` types, and mentioned in sec. 1.

Table 3. Relative differences of the BETs for our *old* benchmarks (as in our preliminary research), now in Program 3-old, to the BETs of our standard benchmarks (from Table 1).

$\overline{\Delta t}_{BET,rel}$ *(Prog. 3-old / Prog. 1)/%*						
Arr. typ.	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%
`short`	−5.0%	+37.3%	−38.0%	−13.3%	+47.2%	+0.1%
`int`	−12.2%	+46.1%	−36.9%	−12.6%	+43.6%	0.0%
`llint`	+40.6%	−14.2%	−0.3%	+26.9%	−10.3%	−0.6%
`float`	−14.6%	+68.0%	−0.2%	−13.7%	+42.6%	0.2%
`double`	−16.4%	+55.1%	−0.5%	−13.1%	+34.2%	+1.2%

Table 4. Relative differences of the BETs for our standard benchmarks (Program 1) compiled to the x64 machine code from the BETs of the same benchmarks compiled as x86 code, in Table 1.

$\overline{\Delta t}_{BET,rel}$ *(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%

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

The 64-bit, x64 machine code did not result in much faster executions (cf. §3.2.3). By inspecting Table 4, we see the 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, because one would expect that the x64 compilation would improve at least the executions for the compatible, 64-bit data types. In addition, the A.3, 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 are in the range from around 10% up to 25%. The B-type benchmarks' BETs are from 10% shorter 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.

Finally, 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, it turns out they are significantly faster (from 10% to 12%) only for the B.3 benchmarks (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 it was 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 our all test programs, with a few smaller 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 the other. 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

The first intention of this research 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 to them via pointers. However, this paper shows that the answer to this question is neither unique nor simple, and even less easy to explain.

Here we upgraded the benchmarks from our ad-hoc measurements from a decade ago, in which the array-element assignment statements were kept as simple as possible and had 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 included also 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 (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 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 disassembly in assembly language. These benchmarks can be regarded as the starting ones upon which the future methodology will be built. In the rest of section 3, we have explained the machine instructions of the compiled code and discussed a few pitfalls that we had encountered in this research and had to deal with. In doing that, we could notice the consequences of the non-orthogonality of Intel's x86 instruction set, in the sense that not all addressing modes are allowed with all operations, which can result in changes in how the source code can and will be compiled (§3.2).

However, despite all the analysis, explaining and understanding many varieties and peculiarities of the benchmark execution times (BETs) in section 4 was everything but an easy task. That 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.

By looking at the results of this research, there is no simple conclusion and no unique best choice. The best method largely depends on the type of the assignment/ operation in the loop body and on the array (data) type. 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 benchmarks are rather non-general because they have constants in the crucial parts of their loops. Besides that, the advantage occurs for only the first three integer types (Table 3).

For our standard benchmarks, with the assignment-only statements in the loop bodies, we have given the concrete advice based on the execution times in Table 1, summarizing them in the (bulleted) list in §4.3.1. For the modified benchmarks, with addition and assignment, the conclusion based on the results in Table 2 is summarized in the list in §4.3.2. Interpreting these results the other way around, we could say that using the indices can be as good as anything else if we primarily do iterative summations and if a few percentages better performance 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 an exception. Etc.

Though, one conclusion is rather simple: the x64 code does not improve the benchmark performance significantly, not even for the 64-bit types. Quite the contrary, from Table 4 we see 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 shortest execution times.

This conclusion, and especially the last piece of advice, also hints us at a few suggestions for future work. First, the additional benchmarks should be written in a way that could not be trivialized by the compiler optimizations, despite the fact that 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 platforms.

6 References

C++ reference (2022). `std::chrono` library, `high_resolution_clock` class. Web contents: https://en.cppreference.com/w/cpp/chrono/high_resolution_clock.

Evans, D (2022). *x86 Assembly Guide*. Web contents: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html.

Intel (2022-1). *View disassembly code in the Visual Studio debugger.* Web contents: https://docs.microsoft.com/en-us/visualstudio/debugger/how-to-use-the-disassembly-window?view=vs-2022

Intel (2022-2). *Intel® 64 and IA-32 Architectures Software Developer Manuals*. Web contents: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

Kernighan, B. W., & Ritchie D. M. (1978, 1988). *The C Programming Language (1st and 2nd Editions, respectively), Englewood Cliffs, NJ: Prentice Hall.*

Logozar, R. (2012-1). Recursive and Nonrecursive Traversal Algorithms for Dynamically Created Binary Trees. *Computer Technology and Application (David Publishing)*, 3**(**5**)**, 374-382.

Logozar, R. (2012-2). Algorithms and Data Structures for the Modeling of Dynamical Systems by Means of Stochastic Finite Automata. *Technical Gazette* (*Tehnički vjesnik*), 19(2), 227-242.

Microsoft, MS (2022-1). Documentation: SYSTEM-TIME structure (miniwinbase.h). Web contents: https://docs.microsoft.com/en-us/windows/win32/api/minwinbase/ns-minwinbase-systemtime

Microsoft, MS (2022-2). Documentation: Acquiring high-resolution time stamps. Web contents: https://docs. microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps?redirectedfrom=MSDN#resolution-precision-accuracy-and-stability

Stroustrup, B. (1997), *The C++ Programming Language (3rd Edition).* Murray Hill, New Jersey: AT&T Labs.