Inverted Run-Time Behavior of Classic Data Structures on Modern Microprocessors: Technical Background and Performance Guidelines

Michael Bogner, Andreas Hofer, Maria Hronek, Franz Wiesinger

University of Applied Sciences Upper Austria
Department of Embedded Systems Engineering
Hagenberg, Austria
Email: {michael.bogner, andreas.hofer, maria.hronek, franz.wiesinger}@fh-hagenberg.at

***Abstract*—Classic data structures, such as vectors and lists are used for storage and organization of data. Certain basic operations have a specified run-time behavior, which is essentially influenced by the choice of the data structure. However, further advances in the development of modern microprocessors have achieved sophisticated optimizations in hardware. These optimizations affect the run-time behavior of certain operations, which further affects the choice of data structures. This paper presents the results of our research activities focused on the impacts of these changed conditions. We selected various algorithms and operations frequently used in todays software development. Remarkable differences and modified characteristics will be discussed. The performances of both selected data structures, namely vector and list, have been determined empirically using the programming language C++. The results are interpreted and discussed in terms of run-time complexity and modern processor development.**

Keywords–performance patterns; sequence container; run-time complexity; modern microprocessors;

I. INTRODUCTION

The performance of microprocessors increased greatly in recent years. This was possible mainly because of sophisticated optimizations in processor architecture, achieved by modern processor development. One substantial reason for the high processing speed of modern Central Processing Units (CPUs) is the hierarchy of various storage levels of cache memory on the processor die. But, in our consideration, also the main memory is important. It is critical to organize data as efficiently as possible in the main memory in order to get maximum performance. Different container data structures allow us to select the appropriate organization. However, not all container types benefit in the same way of modern processor architecture, which leads to different performance gains. Under these changed conditions, current approaches need to be reconsidered.

For the comparison in this paper, we have selected data structures from the Standard Template Library (STL) of the widely used C++ programming language. As performance is a key aspect, we decided to avoid languages which have managed runtime features or rely on virtual machine support, or use Just-In-Time compilation (JIT) and garbage collection. C++ compiles to native code, is platform independent and comes with efficient container implementations of the STL, which substantially reduces interfering side effects.

The selected containers are std::vector and std::list. Both are basic sequence containers, but as generally known, rely on completely different implementations. The std::vector uses a strict byte-sequence and therefore guarantees a contiguous storage space in memory [1]. In contrast, the std::list is a doubly linked list which represents the simplest form of a graph-based data structure, except for the rarely used single linked list. This is a significant difference for the microprocessor, and this paper analyses right this aspect. Advanced data structures like sets and trees are variants of these basic implementation types. Usually, they use hash functions or other optimizations to gain algorithmic benefits. But these algorithmic optimizations are not directly related to the performance of the microprocessor.

The theory teaches that each data structure has its advantages depending on the scenario. Operations such as inserting, deleting, or accessing elements have a proven run-time behavior, which is described by the asymptotic run-time complexity or the big $\mathcal{O}$-notation [2]. This notation makes it possible to specify the run-time as a function of the problem size. Or in other words algorithms can be classified by how they respond to changes in the container size. In software development, this classification essentially influences the choice of data structure.

For the reasons mentioned above, this paper investigates whether classical selection criteria for list and vector, described by the $\mathcal{O}$-notation are still valid or not. In order to achieve the objective, prototype implementations on both data structures were evaluated. The performances were determined empirically on a common workstation; for details, see Section III-B. To measure the run-time and the run-time behavior, various algorithms and operations were selected, which are frequently used in today software development. The results were interpreted and discussed in terms of run-time complexity and modern processor development. Finally, the main rules that can be held responsible for the results are filtered out, to get general performance patterns. These performance patterns represent some guidelines for todays software development on modern microprocessors.

Already Niklaus Wirth noted the importance of data structures in order to create effective applications [3], as data organization is highly relevant. There are quite a lot text books and papers introducing elementary and advanced data structures and corresponding algorithms like [4], including its $\mathcal{O}$-notation. Professional usage of the C++ standard template library containers and their performance guarantees are covered by Musser et al. [5]. To analyse already existing applications, Liu and Rus [6] present a tool for detecting poor data structure selection in C++ programs, which gives a context sensitive performance advice. An automated tool to

 ISBN: 978-1-61208-367-4

generate cost models of given data structures is introduced by Jung [7]. After a training phase to understand the effect of architectural behavior, the statistical data are then fed into a machine learning model which tries to determine the optimal data structure. Its vision is to build it into data structure libraries so that the compiler can automatically select the best implementation.

The paper has the following structure: The first Section presents a short introduction. Section II describes the development of modern processor architectures and why that prefers specific data structures. Section III presents the methodological approach of the test procedure. Also the selected performance tests are presented. The most distinctive test results are presented and analyzed in Section IV. Section V shows the main rules that can be held responsible for the results, whereas the final thoughts are presented in Section VI.

II. MODERN PROCESSOR ARCHITECTURES

This section gives important background information that can be made responsible for the special effects discussed in this paper. These include a short introduction of the properties of modern processor architectures and their effect on classic data structures. Especially, the memory architecture and how modern CPUs optimize memory accesses are points which are addressed, because almost exclusively array data structures benefit from certain optimizations of modern microprocessors, as we see below. Particularly, in view that the memory access is a growing bottleneck, such things can be critical and lead to significant shifts in the performance analysis.

According to Moore's Law, the integration density of transistors on integrated circuits doubling every two years [8] [9], which made more complex and fast CPUs possible. Thanks to a better understanding of the architecture, the Instruction Level Parallelism (ILP) and increasing clock speeds allows to raise the processor performance considerably long time. Described by ILP, the executed Instructions Per Clock cycle (IPC) of the processor were increased. This was achieved by techniques such as pipelining, super scalarity, out-of-order-execution, branch-prediction or speculative execution [10].

One of the biggest challenges during this development was to design memory systems that can provide the processor fast enough with data. Because the increase in speed of modern processors is not accompanied by a corresponding acceleration of the memory systems. This means that memory access is quite slow in relation to processor performance. Figure 1 shows the development of memory and processor performance since 1980 - note the logarithmic scaling in this diagram to be able to display the large gap in this development. The memory line starts with 64 KiB DRAM 1980 and reached an annual latency increase of factor 1.07. The speed increase at the processors reached factor 1.25 until 1986, 1.52 until 2004 and 1.2 after 2004 [11].

Not only the latency, but also the bandwidth to the memory system is important for the CPU. These two points stay in strong conflict to each other. Therefore, a technique for increasing the memory bandwidth often results in an increase of latency, and vice versa. The higher the speed of processors grows, the harder it is to realize a memory, which can provide

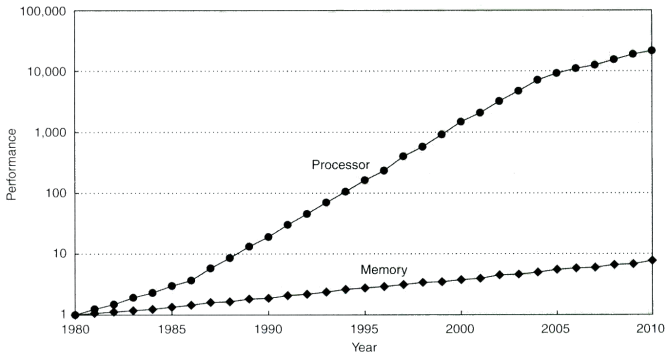

Figure 1. Performance of processors and latency of the memory plotted over time [11].

data fast enough in a few clock cycles. Therefore, the memory system was an increasing bottleneck [12].

To minimize this problem at least, modern processors have various cache levels on die. The purpose of the cache is to take the last used memory words, whereby the new access to them is greatly accelerated. If a sufficiently large amount of the required data is present in the cache, the effective memory latency shrinks enormously. This advantage is tried to maximize with several cache levels [12].

To reach their target, caches feature a variety of optimization; many of them use the memory address of the corresponding data (address locality): The so-called spatial locality refers to the observation that memory locations, that are numerically similar to locations which were accessed recently, will be accessed in the near future with increased probability. This property is exploited by caches reading more data than requested, in the assumption to predicting future accesses. Such optimizations are called prefetching, as data will be already prefetched from the main memory [12]. In terms of the performance tests in this paper, it should be noted that almost exclusively array data structures benefit from these optimizations, because of their contiguous memory order. For lists with a scattered memory order, such optimizations are nearly useless.

III. METHODOLOGICAL APPROACH

This section shows the methodological approach performing the test. The selected tests are described and their purpose is explained. Also, the test system is presented with the hardware and software base. Finally, criteria of the test procedure are determined.

A. Selected performance tests

In principle, we selected various algorithms and operations frequently used in todays software development. We tried to figure out the respective advantages and disadvantages of the two data structures on an modern microprocessor and want to illustrate if classical selection criteria for list and vector are still valid. In Table I, the selected performance tests are presented. The first tests will cover basic operations, such as inserting or deleting elements. Subsequent test cases also check moving, comparing, swapping or sorting items in the containers - in

 ISBN: 978-1-61208-367-4

various combinations and executions. Not all test cases modify the size of the data structures; some only change the order of the data within the containers or are read-only tests. Details are noted in Table I.

TABLE I. SELECTED PERFORMANCE TESTS: EXECUTED TEST CASES AND THEIR DESCRIPTION AND PURPOSE

Test case	Description
Filling data structures	This is one of the most common operations in general. The elements in each of the newly declared containers are inserted at the back. In addition, also a vector is tested which gets the final size communicated via vector::reserve.
Clearing data structures	It is tested how quickly a filled container can be cleared completely. Since vector::clear does not change the capacity, for a fair comparison a new container is created using vector::swap.
Insert front	Also the insertion of elements on other positions are meaningful test scenarios. In this test case, additional elements will be inserted at the front of already filled containers.
Insert middle	It is measured how long it takes to insert additional elements in the middle of already filled data structures. The respective advantages and disadvantages of the two test candidates can be very well shown in this test case.
Insert sorted	In contrast to insert middle, in this test case the insertion point must be found first, so the elements in the data structures have to be accessed. The test starts with empty containers.
Reversing data structures	This test case reverses the order of the values in the containers. The first value becomes the last one, the second value the last but one etc. Therefore, the last value becomes the first value of the reversed container. It is shown how efficient elements in the containers can be swapped.
Is sorted	It is checked whether sorted containers are actually sorted. It just depends on how fast the data structures can be run through and accessed, making this text case clearly different from the previous.
Calculation of the arithmetic mean	The arithmetic mean is calculated over the containers. Similar to the previous point it must be iterated through the data structures. Not merely elements must be compared with each other, but also arithmetic operations occur.
Delete all occurrences of a number	In this test case, the deletion of specific numbers is measured. This process is repeated with all numerical values until the data structures are completely empty. Also different implementation variants are tested using the vector, to show the respective advantages and disadvantages.
Stable sort	A frequently occurring operation is also the sorting of data structures, which is represented by this test case. As the sorting algorithm of the std::list is stable, the vector uses also a stable sorting algorithm for a fair comparison.
Delete all duplicates	Every container is filled ten times in a row with the same number. It will be measured how long it takes to delete all duplicates. With one run though the data structures, the size is reduced to a tenth of its original value, what this test case clearly differ from the others.
Double each element	Each element in the containers is placed next to the current position again. Thereby the container size is doubling. The advantages and disadvantages of the vector compared to the list can be illustrated nicely in this test case.

B. Test system

The performance tests took place on the following system:

- Intel® Core™ Processor i5-3570K, 4 x 3.40GHz
- Corsair® Memory 16 GiB DDR3-1333 CL9
- Intel® Media Series DH77EB Mainboard
- Samsung® SSD 830 Series 256GB, SATA 6Gb/s
- Microsoft® Windows™ 7 Home Premium 64-bit with Service Pack 1 (March 2013)

The chosen test system represents a common workstation, no high-end device or special hardware. This should demonstrate the general validity of the test results on widespread available systems. The Intel® Core™ family has not changed significantly in those points relevant for the test, eg. from used Ivy Bridge model to the current Haswell architecture. The respective tests were implemented using Visual Studio 2010 Ultimate with Service Pack 1 and was compiled with compiler version 16.00.40219.01. Furthermore, the x64 version was used as a release build. The default settings of Visual Studio were used, with optimization level "O2 maximize speed".

C. Criteria of the test procedure

In order to ensure meaningful results, the following criteria have been defined:

- It was ensured that there is no main memory overflow. Because outsourcing of data on the hard drive would lead to significant performance degradation. The containers are small enough to find place in the main memory in any case.
- Dynamic frequency scaling (Enhanced Intel® SpeedStep® Technology (EIST) and Intel® Turbo Boost Technology) has been disabled. Therefore, the processor is running with the base clock of 3.4 GHz during the tests.
- Every test result has been repeated several times and the arithmetic mean has been extracted from the times of the measurements. This avoid measurement errors and reduce the possible impact of background processes from the operating system. Details will be shown in Section IV.
- To evaluate the run time behavior, five different container sizes were selected for each test case.
- The selected value type of each data structure is integer. Therefore, each element represents a four-byte signed value.

IV. PRESENTATION AND ANALYSIS OF THE RESULTS

Three test cases out of the twelve shown in section III-A are now presented and analyzed in detail. These most distinctive test cases, which provide particularly remarkable results are "Filling data structures", "Insert sorted" and "Stable sort". The results were interpreted and discussed in terms of run-time complexity and modern processor development.

A. Filling data structures

In this test case, the elements in each of the newly declared containers are inserted at the back. In addition, also a vector is tested which gets the final size communicated via vector::reserve, avoiding typical resize operations. The respective container sizes and the results are shown in Figure 2. In order to ensure sufficient runtime, each test ran 100 times and run times were summed up.

The vector dominates this comparison against the list. Beyond that the vector that has reserved all required memory before the measurement is nearly three times as fast as the normal vector. Both containers allow the insertion at the end with the run-time complexity of $\mathcal{O}(1)$, but there are significant differences in detail: the vector allocates memory always for multiple elements, to avoid of requesting new memory every

 ISBN: 978-1-61208-367-4

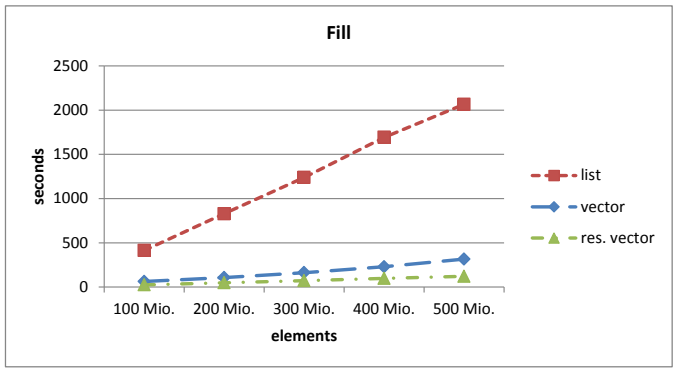

Figure 2. Filling data structures: The elements are inserted at the back. The res. vector gets the final size communicated via vector::reserve.

time. But if the vector is stored in a too small memory area, the entire container must be copied into a larger storage area, because of the contiguous memory order. Of course, the reserved vector does not have this problem, because it gets the final size at the very beginning.

For the list, it does not matter where the elements are stored in the memory. Because it is a doubly linked list with a tail pointer, it has also direct access to the last element. However, due to the link pointers, much more memory must be requested, and also the correct connectivity of the list nodes must be ensured. Much more salient is that the list must request for each node separately new memory. There must be permanently found free space on the heap, which represents a considerable overhead. This explains why the vector dominates the list.

B. *Insert sorted*

In this test case, both containers insert the same random numbers in sorted order. Before a new number can be inserted, the insertion position must be found. The range of these numbers moves between zero to 10,000. At the beginning, both data structures are empty. Figure 3 shows the results with different container sizes for vector and list.

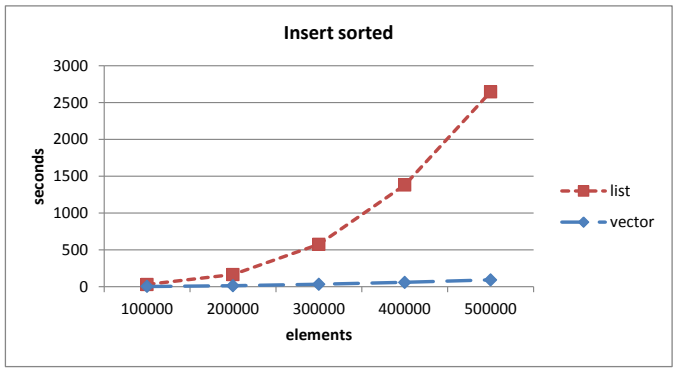

Figure 3. Insert sorted: list and vector in each case insert the same random numbers in sorted order.

It can be seen that the vector dominates this comparison very clearly. Because of its direct access to every element, the vector could use a binary search algorithm to find in $\mathcal{O}(log(n))$ the insertion position. However, the list must be run through linearly element by element. But this is not the only reason for the big advantage of the vector. Figure 4 shows a vector comparison, once with linear and once with binary search algorithm. The binary search algorithm is of course faster, but the difference is not large enough to justify the enormous advantage over the list.

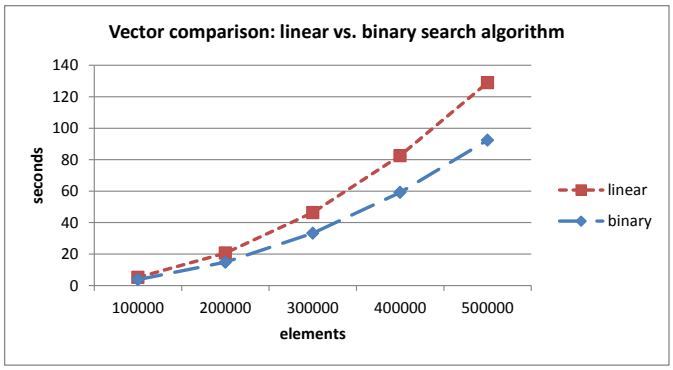

Figure 4. Vector comparison linear vs. binary search algorithm: The vector is with both algorithms much faster than the list.

Since the actual insertion of elements in a list takes hardly any time, finding the insertion position and allocating a new list node are responsible for the high run-time of the list. When the list must be linearly traversed, the address of the successor node can only be determined if the current list node is already loaded from the memory. Because of this data dependency, neither the compiler, nor the CPU has any opportunities to optimize. This explains the poor performance of the list.

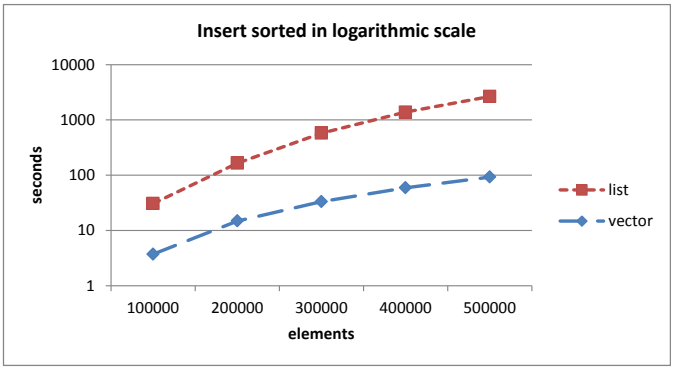

Figure 5. Insert sorted in logarithmic scale: In addition to the better run-time, also the better run-time behavior of the vector can be seen.

On the other side, the vector can be very well optimized thanks to its well-defined data order in the memory. It must be shifted on average half of the container on every insertion, but this takes in relation significantly less time than the list requires finding the insertion position and allocating a new list node. Figure 5 shows the same results as in Figure 3 in logarithmic scale. The vector needs $\mathcal{O}(log(n))$ to find the insertion position and $\mathcal{O}(n)$ to insert elements within the container. Although the list find the insertion position in $\mathcal{O}(n)$ and insert elements within the container in $\mathcal{O}(1)$, the vector has not only the better run-time, but also the the better run-time behavior.

 ISBN: 978-1-61208-367-4

C. Stable sort

In this test case, it is measured 1,000 times to sort the data structures. Both containers will be filled with the same random numbers in the range of zero to 10,000. Since list::sort is a stable function [13], for a fair comparison std::stable_sort is used for the vector. Both functions guarantee a time complexity of $\mathcal{O}(n * log(n))$ [14]. Figure 6 shows the results of this comparison.

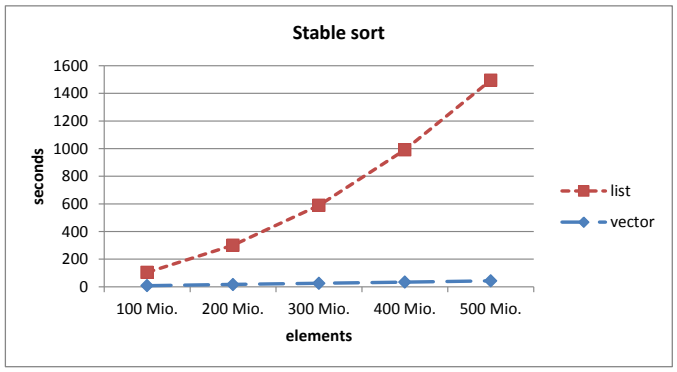

Figure 6. Stable sort: The same random numbers are sorted stable. It is clearly seen that the vector is in advantage, thanks to direct access and compact data.

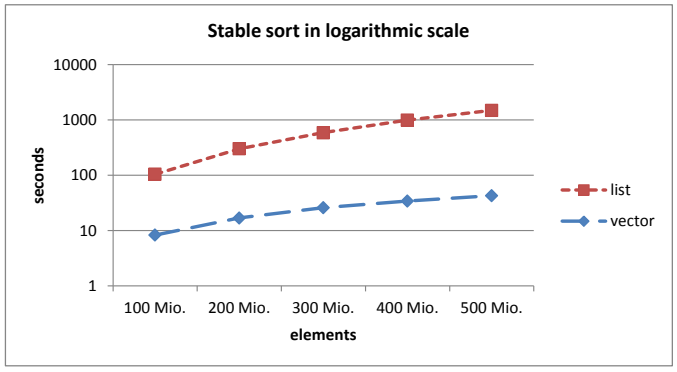

Figure 7. Stable sort in logarithmic scale: despite identical time complexity, the better run-time behavior of the vector can be seen.

Again, the vector dominates this test relatively clear. The direct access on each element in $\mathcal{O}(1)$ and the much more compact data structure of the vector have a positive effect again. Although both sorting algorithms guarantee a time complexity of $\mathcal{O}(n * log(n))$, the vector also has a better run-time behavior, as shown in Figure 7. The $\mathcal{O}$-notation only represents the worst case for the growth of the running time, which does not have to occur.

V. Evaluation

On the basis of theoretical considerations of vector and list, most people would think that it depends on the specific test case which container dominates which test. The run-time complexity of the data structures is quite equal in most of the used tests. But as the results show, a significant advantage of the vector can be seen. The vector dominates all tests with a single exception: ”Insert front”. There, the list has a clear advantage, because no element must be shifted. The smallest differences in run-time and run-time behavior show the test cases ”Insert Middle”, ”Is sorted” and ”Calculation of the arithmetic mean”. But even there the vector is at least twice as fast as the list, thanks to the direct access (”Insert Middle”) and the faster linear traversing (other two test cases). In the test cases ”Double each element”, ”Delete all occurrences of a number” and ”Delete all Duplicates” it is important to consider the characteristics of the vector. Linear traversing and the following insertion or deletion leads to unnecessarily high running times. But thanks to the direct access, elements could be doubled or deleted within the container with very less shift operations, which leads to a significant domination of the vector over the list. At the test cases ”Clearing data structures” and ”Reversing data structures” the vector is approximately ten times faster than the list. This is possible because of the comprehensible storage area of the vector, all elements can be cleared at once and the elements within the container can be easily swapped. The run-time behavior is quite equal in ”Reversing data structures” but at ”Clearing data structures” the vector has also the better run-time behavior.

But, what are the reasons for this clear result under the given test conditions? In this section, the main rules are filtered out from the amount of data that can be held responsible for these results. We have worked out four performance patterns representing some guidelines for todays software development on modern microprocessors.

1) **Linear traversing the data structures:** It is found across all tests that linear traversing of the data structures leads to significant differences in the duration time between vector and list. The vector benefits from the consecutive order of the elements in memory. This leads firstly to a maximum utilization of the limited caches. On the other hand it can be very well optimized, for example when elements are loaded speculatively already in advance. Such optimizations are becoming increasingly important, because main memory is becoming more and more slowly in relation to processor performance.
The elements of the list are scattered in memory, so they must be found costly and the memory access is poorly predictable. The address of the successor node can only be determined, if the current node has been read from the memory. Through this data dependency, the access to a list node can hardly be optimized, pre-loading data from memory (prefetching) is often impossible. Therefore, it takes far too long to go through a list.

2) **Access to items:** Often, a data structure should not be linear iterated, instead it must be random accessed on particular elements. The vector benefits from being able to access any element in constant very short time. This benefit allows to more than compensating other disadvantages. For example, when elements within a data structure should be inserted, deleted or swapped. The vector could use a binary search algorithm to find a specific element within a sorted container and also benefits from his direct access at sorting or reversing the container. The list does not have this advantage; direct access is only at the beginning and the end of the list possible. For any other element, the list must be

 ISBN: 978-1-61208-367-4

linearly traversed starting from the ends - with the same problems as shown previously.

3) **Inserting and deleting elements:** The insertion and deletion of elements at different positions of the data structures shows an ambivalent picture. At the end of the container the vector allows a very fast insertion and deletion of elements. The vector allocates memory always for multiple elements, to avoid requesting new memory every time. If the vector is stored in a too small memory area, the entire container must be copied into a larger storage area, because of the contiguous memory order. It is also possible to tell the vector the final container size at the very beginning, so all memory is requested directly whereby elements can be inserted even faster. When elements should be deleted at the end, the vector needs not to free any memory or iterate through the container, instead the elements can simply be cutted off, without changing the capacity of the container.
The list, however, must be run through in any case, with the exception of the first and the last node. Also, every time a new element is inserted or deleted, the list must request new memory or free used memory on the heap, which represents a considerable overhead. In addition, because of the administrative data, significantly more memory is used and the correct connectivity of the list nodes has to be ensured. Therefore the list needs significantly more time at the end of the container for such operations.
But the farther away from the end of the container elements should be inserted or deleted, the sooner the list has the advantage. The structure (interlinking) of the list has to be changed only locally. The vector must move a substantial number of elements when inserting or deleting, depending on the distance to the end of the container. This is an expensive operation. Therefore, insertion or deletion at the beginning of the vector should be avoided if possible. Usually this problem could be avoided or at least alleviated by clever optimizations or simply by using std::deque.

4) **Memory and cache utilization:** An aspect that should also be addressed is the high memory consumption of the list. The random order of the list nodes in memory leads to a high share of administrative data. At least two pointers need to be stored per element in a doubly linked list. What leads to 16 bytes per element management data in a 64-bit application. If there is not enough main memory available, it would lead to a significant drop in performance, because data must be outsourced on the hard drive.
Beyond that, this large administrative data make the limited caches ineffective, because more overhead and less useful data are stored. The random order of the list nodes leads also to a high number of load operations from main memory, because in the worst case every list node must be loaded separately. On the other side the vector requires very little administrative data and therefore allows storing data very compact. This allows utilizing the main memory and the limited caches best. And because of the contiguous storage area, multiple vector elements can be loaded at once from the main memory.

VI. Conclusion and Future Work

Summarizing, it can be said that modern microprocessors show a quite different run-time behavior for certain operations than one would expect looking at the corresponding $\mathcal{O}$-notation. In the mentioned cases, it is simply misleading following the $\mathcal{O}$-notation, which finally results in low performance of the application.

Certain optimizations of modern microprocessors prefer data structures with a coherent storage area. For lists, optimizations such as pre-fetching algorithms, are nearly useless, because of the data dependency of the link-pointer. The address of the successor node can only be determined if the current list node is already loaded from the memory, which makes it nearly impossible for modern microprocessor architectures to optimize the access to subsequent data. Against this background and the increasingly limited memory system, expensive memory accesses should be designed as predictable as possible and data should be kept compact to utilize the limited cache best. Such challenges prefer data structures with a coherent storage area.

It turns out that classical selection criteria for list and vector have been undermined by modern processor development in some way. So far, the insertion or deletion of elements within the amount of data was a clear domain of the list, since no shift operations are necessary. Today, even the shifting of the vector elements up to a specific position is more efficient than the linear iteration through the list to find the insertion position. If one takes into account certain characteristics of the vector, such data structures should clearly be preferred.

In the future, the performance of CPUs will still increase and additional potential for optimization will continue to prefer data structures with a comprehensible storage area. The so-called prefetching, the speculative load of data from memory, brings an enormous advantage for the vector. The data dependency of the list, which allows neither the CPU nor the compiler to optimize usefully, is a serious problem. This should be considered in software development.

Future work will evaluate additional data structures to get a more comprehensive picture about the run-time and the run-time behavior of different data structures on modern microprocessors.

References

[1] G. Pomberger and H. Dobler, Algorithms and Data Structures - A Systematic Introduction to Programming. Pearson Studium, 2008.

[2] D. E. Knuth, “Big omicron and big omega and big theta,” SIGACT News, vol. 8, no. 2, Apr. 1976, pp. 18–24.

[3] N. Wirth, Algorithms + Data Structures = Programs. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1978.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. MIT Press, 2009.

[5] D. R. Musser, G. J. Derge, and A. Saini, STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library, 2nd ed. Addison-Wesley, 2009.

 ISBN: 978-1-61208-367-4

[6] L. Liu and S. Rus, “Perflint: A context sensitive performance advisor for c++ programs,” in Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, Seattle, WA, 2009, pp. 265–274.

[7] C. Jung, “Effective techniques for understanding and improving data structure usage,” Ph.D. dissertation, Georgia Institute of Technology, 2013.

[8] G. E. Moore, “Cramming more components onto integrated circuits,” Proceedings of the IEEE, vol. 86, no. 1, 1998, pp. 82–85.

[9] G. E. Moore, “Progress in digital integrated electronics,” Electron Devices Meeting, 1975 International, vol. 21, 1975, pp. 11–13.

[10] J. L. Hennessy and D. A. Patterson, Computer Organization and Design - The Hardware / Software Interface, 5th ed. Morgan Kaufmann, 2013.

[11] J. L. Hennessy and D. A. Patterson, Computer Architecture - A Quantitative Approach, 5th ed. Morgan Kaufmann, 2011.

[12] A. Tanenbaum, Computer Architecture: Structures - Concepts - Basics, 5th ed. Pearson Studium, 2006.

[13] B. Stroustrup, The C++ Programming Language, 4th ed. Addison-Wesley, 2013.

[14] U. Breymann, C++ - Introduction and professional Programming, 9th ed. Carl Hansen, 2007.

 ISBN: 978-1-61208-367-4