Energy-efficient application programming for green cloud computing

Tomáš Zeman[1], Jan Fesl[2], Ludvík Friebel[3]

DOI: 10.32725/978-80-7394-976-1.36

Abstract: Green Cloud Computing is a very interesting area that deals with different ways to reduce the energy consumption of clouds and data centres. Software solutions (native applications or those running inside containers) whose optimization (especially at the binary code level) can achieve a significant increase in computational performance or a reduction in computational time and thus directly reduce power consumption have a significant impact on the power consumption in these environments. In our study, we focused on the optimization of programming code in the C++ programming language, both in terms of the syntactic constructs of the programming language and the code generator itself. Our findings show that the difference in the efficiency of the resulting binary form of the program can be as much as tens of percent lower in terms of energy consumption.

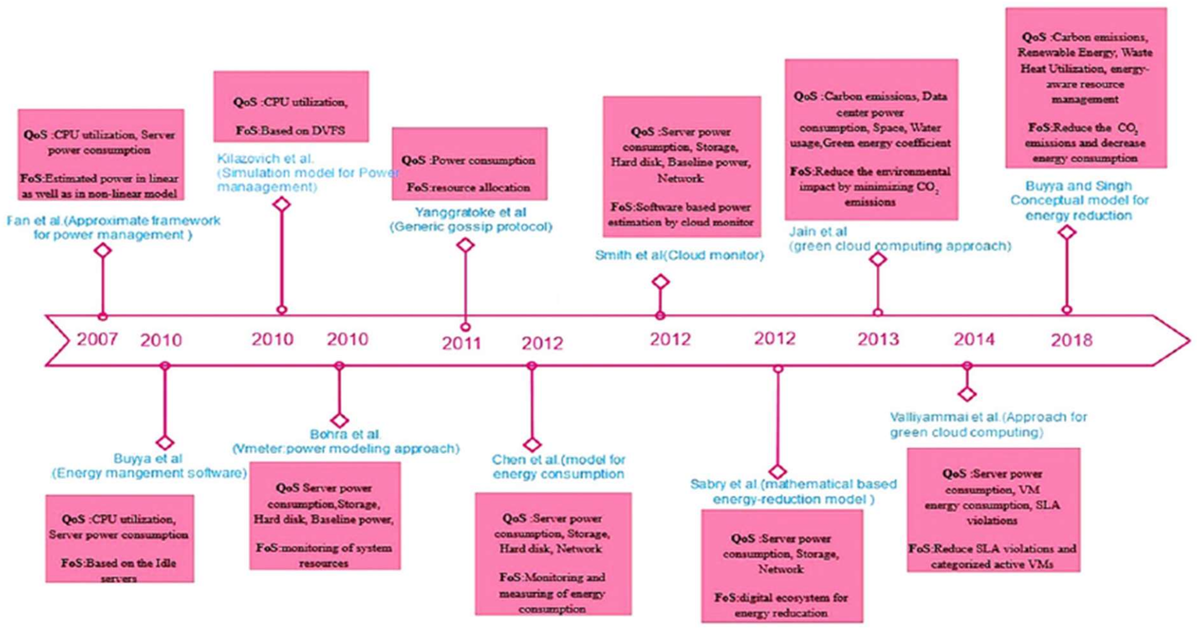
Keywords: Cloud computing, code, optimization, energy, efficiency, green computing
JEL Classification: C69, C80, P18, Q55

1 Introduction and motivation

Since cloud infrastructures consume a huge amount of power, finding ways to reduce this power consumption has been a very hot research topic in recent years (Lefevre, Orgerie, 2010), (Nordman and Berkeley, 2009). Several concepts have been proposed (Younge et al., 2010) that independently are able to reduce power consumption in a dramatic way. Concepts that allow energy reduction are generally divided into hardware (tangible) and software (intangible) (Bharany et al., 2022). Hardware concepts include e.g., solar energy systems, buildings with continuous heat circulation, server rooms with alternative cooling (direct air intake), etc. and software concepts include e.g. orchestration systems for the management or development of energy efficient applications (DEEA). All mentioned concepts, with the exception of DEEA, depend on a specific infrastructure for which they have to be specifically optimized. The evolution of solutions across time (Esmaeilzadeh et al., 2011) can be seen in Figure 1, with hardware-oriented (Khanna et al., 2011), (Mashayekhy et al., 2015) solutions appearing first, and later software-oriented solutions (Ketankumar et al., 2015) and (Singh et al., 2015). A popular solution is the use of methods from the field of artificial intelligence (Chen et al., 2016), or even tracking user activity (Kim et al., 2011), (Lin, 2012).

[1] University of South Bohemia in České Budějovice, Faculty of Science, Department of Informatics, Faculty of Science, Branišovská 31a, 370 05 České Budějovice, Czech Republic
2 University of South Bohemia in České Budějovice, Faculty of Science, Department of Informatics, Faculty of Science, Branišovská 31a, 370 05 České Budějovice, Czech Republic
3 University of South Bohemia in České Budějovice, Faculty of Economics, Department of applied mathematics and informatics, Studentská 13, 370 05 České Budějovice, Czech Republic, ludva@ef.jcu.cz

Figure 1 Evaluation of promising techniques used in Green Computing within the years 2007-2018, source (Bharany et al., 2022)



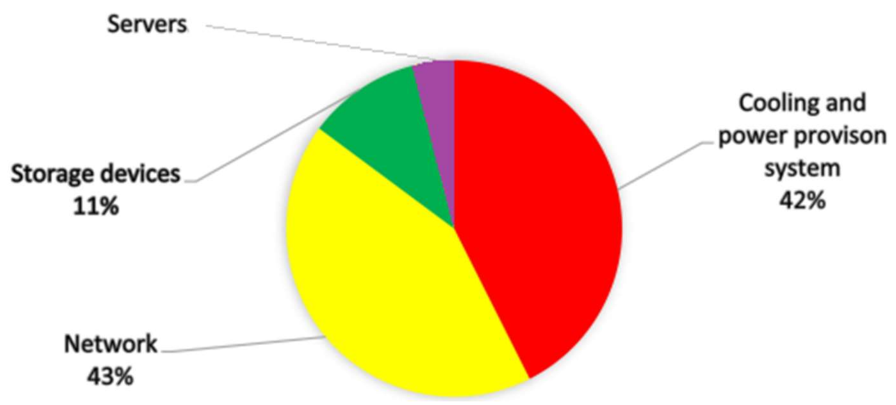
QoS :CPU utilization, Server power consumption
FoS:Estimated power in linear as well as in non-linear model
Fan et al.(Approximate framework for power management)
QoS :CPU utilization,
FoS:Based on DVFS
Kilazovich et al. (Simulation model for Power manaagement)
QoS :Power consumption
FoS:resource allocation
Yanggratoke et al (Generic gossip protocol)
QoS :Server power consumption, Storage, Hard disk, Baseline power, Network
FoS:Software based power estimation by cloud monitor
Smith et al(Cloud monitor)
QoS :Carbon emissions, Data center power consumption, Space, Water usage,Green energy coefficient
FoS:Reduce the environmental impact by minimizing CO_2 emissions
Jain et al (green cloud computing approach)
QoS :Carbon emissions, Renewable Energy, Waste Heat Utilization, energy-aware resource management
FoS:Reduce the CO_2 emissions and decrease energy consumption
Buyya and Singh Conceptual model for energy reduction
2007
2010
2010
2010
2011
2012
2012
2012
2013
2014
2018
Buyya et al (Energy mangement software)
QoS :CPU utilization, Server power consumption
FoS:Based on the Idle servers
Bohra et al. (Vmeter:power modeling approach)
QoS Server power consumption,Storage, Hard disk, Baseline power,
FoS:monitoring of system resources
Chen et al.(model for energy consumption
QoS :Server power consumption, Storage, Hard disk, Network
FoS:Monitoring and measuring of energy consumption
Sabry et al.(mathematical based energy-reduction model)
QoS :Server power consumption, Storage, Network
FoS:digital ecosystem for energy reducation
Valliyammai et al.(Approach for green cloud computing)
QoS :Server power consumption, VM energy consumption, SLA violations
FoS:Reduce SLA violations and categorized active VMs

The development of energy-efficient applications is a trend that has been addressed with varying intensity for many decades and the current global energy crisis makes this trend very important again. For the development and optimization of energy-efficient programs, native compilable programming languages such as C, C++, Go or Rust are particularly suitable. This is because compilers of these programming languages generate direct executable binary forms of programs that are natively executed by concrete processors. The programmer can therefore influence the efficiency of the resulting binary program form by both the design of the application and the use of appropriate program constructs.

2 Current State-of-the-Art

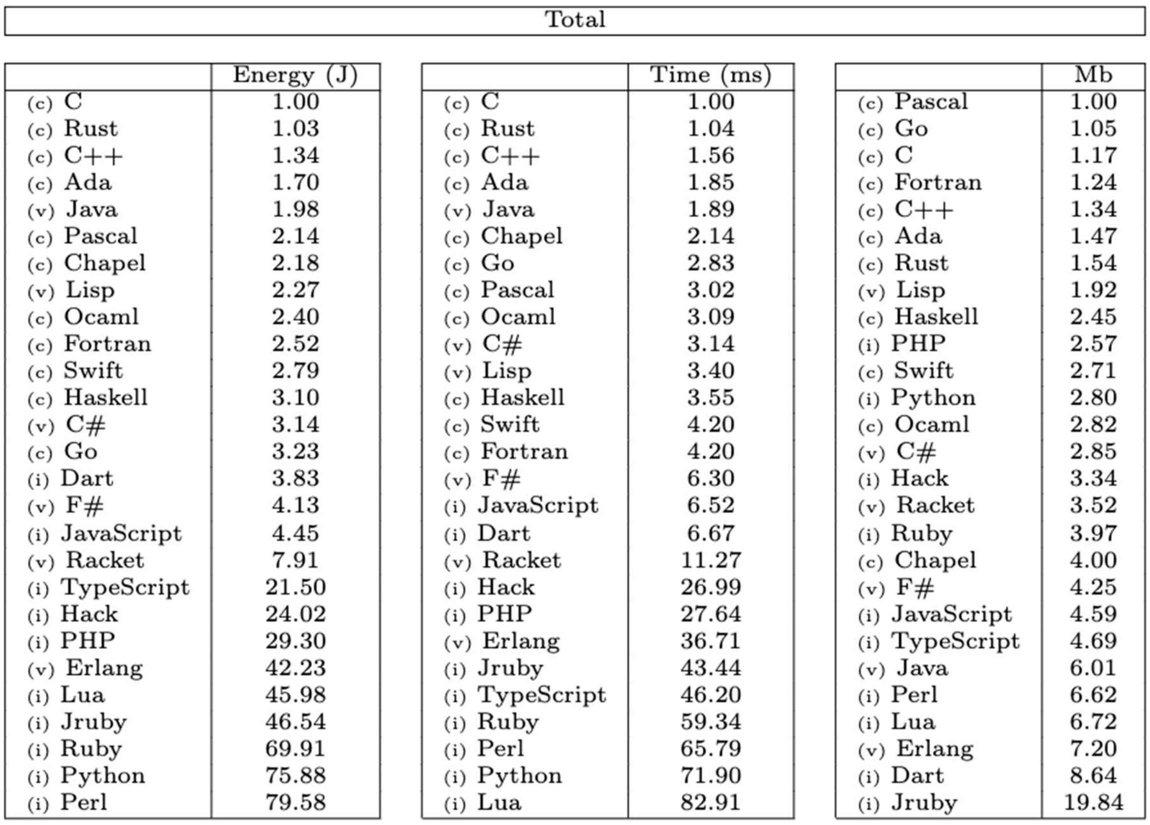
According to (Bharany et al., 2022) the typical total consumption of a data center is shown in Figure 2. A relatively small fraction of energy is left for server operation and reducing consumption through efficient application implementation is particularly important in this area.

Figure 2 Energy consumption graph based on US Datacenters in 2014.



A comprehensive comparison of implementations and efficiency measurements was meant (Pereira et al., 2021). Table 1 contains the specific data regarding the code efficiency measurements. The measurements were oriented on consumed energy [Joule], on the total application execution time [ms], and on the size of used memory [Mb].

Table 1 The energy efficiency comparison of currently most used programming languages



Total					
	Energy (J)		Time (ms)		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Source: (Pereira et al., 2021)

The measurements confirmed the idea that the programming languages which are not interpreted are able to achieve substantially better energy efficiency.

3 Contribution

In our paper, we focused on the optimization of programs written in the C++ programming language of its latest variants and tried the use of advanced programming constructs and experimentally compared the efficiency of the resulting binary code with or without their use. A sample of some optimizations is shown below.

About new tested features

Selected C++20 features (Calandra, 2022) have been analysed and tested, those in themselves have the character that inherently improves performance in specific ways. While conducting the performance tests we measured units for execution time, compile-time, memory usage during translation and size in bytes of the whole translation unit. Those units were always specifically chosen and measured according to character that the given feature in the new standard provides.

Techniques used in the measurement

Created was a custom benchmark for execution efficiency using Chrono library (Microsoft documents, 2022). For our purpose the mentioned library provides very accurate CPU time and it's also very popular among testers. Testing was performed on three different computing setups with those unique operating systems: Windows 10, Ubuntu and macOS. The measurements were also carried out using those three most used compilers: g++, clang++ and MSVC.

Testing each time took place in an isolated, prepared environment to obtain accurate and also unaffected results. Used algorithms were developed and optimized purely for our testing purposes. That means we strive for higher time complexity within a given algorithm. This allows the program to run or compile for several minutes for relatively low input parameters. In this way, we achieve precise measurable results that can be further statistically analysed.

Selected language features

Attributes [[likely]] and [[unlikely]]

Conditions and if statements are generally one of the most frequently used constructs across all programs. New attributes improve performance in evaluating given conditions. Because conditions are so frequently used, influencing evaluation output then makes a significant difference in overall performance.

Consteval and Virtual Constexpr - Compile time evaluation

Compile time evaluation is improved and extended with each edition of the standard. Latest C++ standard brings new two keywords with a useful rich application. Main idea of the whole usage is that the appropriate calculations will be evaluated during the translation time, which will subsequently speed up the runtime of a program.

Modules - a new compile units

Most of C++ projects are using multiple translation units (cppreference.com. Modules, 2022). You can easily separate the interface from its implementation. Up to now, it has only been possible to use header files, but they suffered from a number of inconveniences. Main reason for our deeper exploration is that for many large projects, translation times are often very disproportionate, and order of imported units suffer from unexpected bugs. New modern way called modules eliminates these problems. It basically delivers an improved and much faster solution (Lischner, 2020).

Description of features and measurement principle

Tested attributes [[likely]] and [[unlikely]] allow the compiler, or rather the optimization performed, to modify the generated form so that it executes significantly faster when evaluating conditionals in if-else branching (Stroustrup, 2020). The whole idea behind usage is that conditional branching in very frequent cases is not distributed with uniform probability. In fact, a specific condition is usually evaluated with a certain superiority than others. Described situation does occur frequently, not only in large projects, and the use of this new functionality can result in significant execution time savings. Attributes affect only the speed of the algorithm during program execution, but not its translation time.

Syntax

```
if (condition) [[likely]] {

// code

} else (condition) [[unlikely]] {

 // code

}
```

The principle consists in comparing execution time of algorithms for standards C++17 (fundamentally without attributes) and C++20, both using optimization parameter -O3. An important remark, all algorithms do not dispose of third-party libraries that further affect any aspect of performance. The evaluated results of algorithms are stored in a variable of type volatile to avoid unwanted side effects. In this case, our previously mentioned custom benchmark was used.

The benchmark takes a total of 10 samples per one run and averages them, while ignoring the best and worst results. Testing was carried out for three different input parameters each time, with each parameter being tested thirty times. This means a total collection of 900 possible results. Efficiency is thus assessed based on the measured execution time of both versions.

Consteval and Virtual Constexpr

Since the C++11 standard, which introduced the keyword constexpr, there is a possibility to evaluate functions or variables directly at compile time. The declared constexpr function, however, does not directly guarantee this property to the user with certainty. It is only able to be used in this way under certain conditions and with constant parameters (Fertig, 2021). The ISO C++ committee decided to clearly define this sometimes-unclear notion, and so the keyword consteval was newly introduced, which directly guarantees that the function will necessarily be evaluated at compile time. If this case could not be feasible for certain reasons, the program will not be able to be compiled afterwards. This option again gives us more control over the code. What's also new is that virtual functions can now be evaluated during compilation. (Stroustrup, Sutter, 2022).

Since the C++20 and C++17 standards do not differ in terms of the compile time speed of the created algorithms, we will take advantage of the new extended capabilities of the latter standard to transfer the computations to the compile time and compare them non-competitively with the runtime by using custom benchmark of a C++17 program.

However, these units are not directly comparable to each other, which is not the stated purpose of the measurement. The stated goal is to use a completely new way of speeding up the application which has not been possible yet.

The results are then evaluated in such a way that the calculation of the algorithm in C++20 will always run at a constant speed, so we will measure the compile time using the parameters of the respective compilers. On the other hand, for C++17 we see by what time difference the application in C++20 would be faster in the runtime of the program. Thus, the main idea is to convey motivation by showing how certain calculations transferred to the compile time will effectively speed up an application and as well as to achieve better results. Indeed, in general, for any non-trivial program, there is bound to be a space where such functionality could be deployed at any time.

Modules

The latest standard now comes with a long-awaited and completely new modern and redesigned solution for libraries and other compilation units. Modules thus generally provide a completely new way to work with multiple linked translation units. They also eliminate recurring problems that still have common header files. Modules can be now imported in any order without having concern for macro redefinitions. (Microsoft documents, 2022).

Adding a module starts with the keyword: import module. For the part of the module that is to be exported, for example, a function, namespace, or class we add the keyword export. A module can also easily be divided into several logical partitions, where we then separate the interface from the implementation (Stroustrup, 2018).

The most significant benefit is that once the module is compiled, it is preserved in binary form. Such a module is much faster to process than a header file since the compiler just reuses it at each place it occurs (ModernesCpp.com, 2020).

The comparison in terms of performance differences occurs for programs that use complete compilation either using modules (C++20) or on the other side of header files (C++17). We primarily test compile time, but we also focus on the size in bytes of the compiles themselves and occasionally the size of the memory consumed during compilation. The results of these tests are obtained based on built-in parameters used by compilers and operating systems.

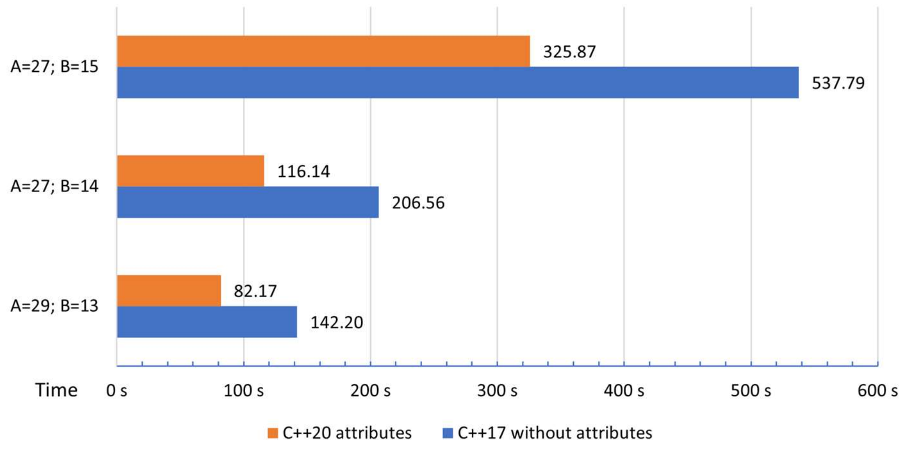
4 Results and Discussion

A total of 8 different programs were created to test the new optimization attributes. Each of them has its own compiled version that belongs to the corresponding C++ standard. The greatest performance increase was observed for the testing algorithm based on the longest common sequence of characters of two strings.

On the other hand, an algorithm based on finding an element in a given range of numbers has a performance increase in lower percentages units, but it is still a certain improvement.

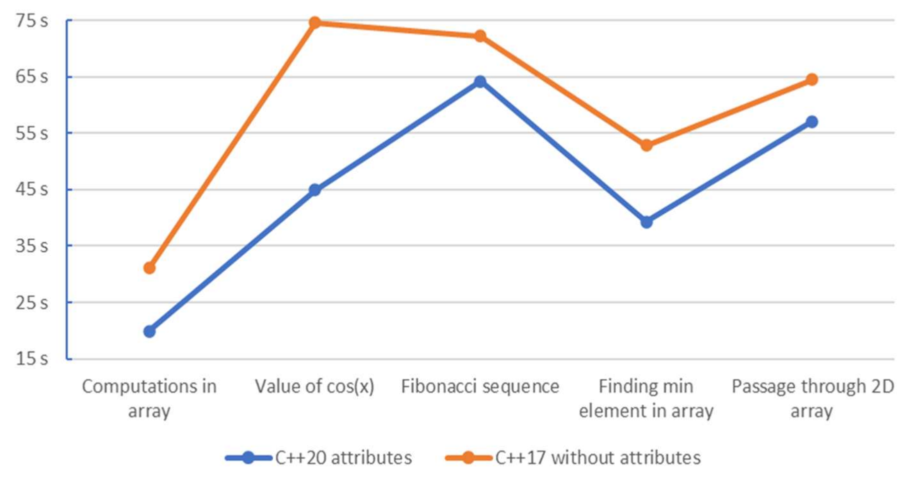
In case of the LCS algorithm, the performance increase reached up to 43.77 % in the best result. In the following graph in Figure 3 we can see the comparison of algorithm speed depending on different input parameters on the left side. Measured execution time in seconds then represents the statistical average of obtained samples.

Figure 3 Results of the LCS algorithm



Now follows the processed and analysed data from five different test algorithms, which have been scored and added to the chart. The bottom part shows names of tested algorithms, and the left part displays the average evaluation time of algorithms across different input parameters.

Figure 4 Comparison of efficiency for created algorithms



Obtained results indicate that there has indeed been a significant performance increase in execution time. A minor remark is that attributes must be properly thought out for a given algorithm, otherwise, we would decrease the performance on the contrary.

We also explored the corresponding assembler outputs to determine the cause of these improvements. Instructions are now modified so that the entire block run is now tailored to the set attributes during evaluation. Thus, the subsequent instruction jump for the evaluation first runs to the location in the conditional instruction register, according to the set attributes. This also rearranges the blocks differently and consequently generates faster code depending on our custom settings.

Advantage of Compile-time computation

In the following content, we discuss the improvement of the program runtime by evaluating the given computations at compile time. For this purpose, the tested programs use new consteval keyword and the others evaluate virtual functions at compile time using also another new virtual constexpr keyword.

The most remarkable result was found on the recursive algorithm based on the well-known Fibonacci sequence. The following Table 2 shows a summary of some selected algorithms. The last two algorithms in the table newly use a possibility of evaluating virtual functions at compile time. The data thus shows required compilation time for a certain input parameter. Run time then displays possible execution time savings of our program.

Table 2 Tested algorithms with compile-time evaluation

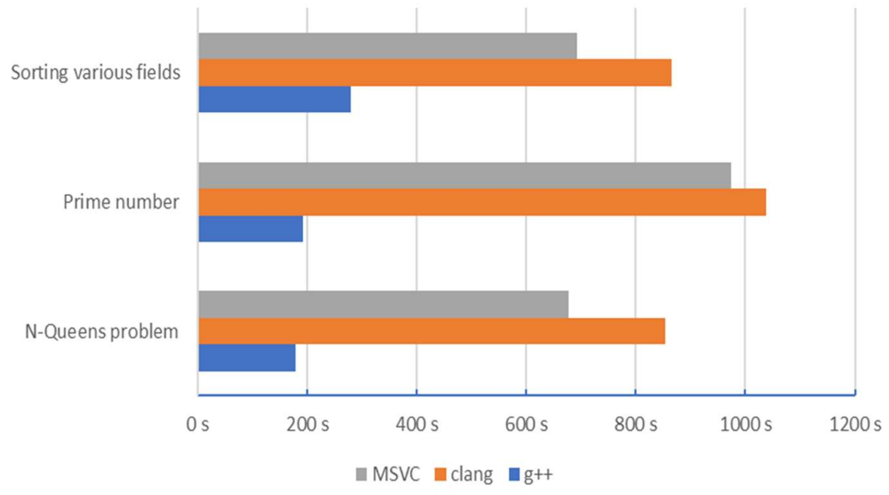
Compiler		**g++**	
Name of algorithm	**Parameter**	**Compile time [min]**	**Run time [s]**
Fibonacci Sequence	N = 46	8.12	12.27
N-Queens problem	DIM = 12	7.47	0.71
Prime numbers	N = 35 K	5.96	1.43
Binomial numbers	N = 31	4.93	0.44

Source: Own processing

From the end user's perspective, this means that an application did not have to evaluate this relatively high computation repeatedly after the build was completed. Imagine all these results on a global scale. On suitably transferred computations, they can speed up the final application without having to make major changes.

The g++ compiler also presented the fastest translation times (tested on a particular setup with OS Windows), often four times faster than clang. Importantly, translations were not affected by optimization parameters that would significantly affect all obtained values.

Figure 5 Compiling speed analysis on Windows PC



Advantage of using these new options is their ease of deployment into a project. After proper testing of an application, these features can be easily and cost-effectively added at any time. Constexpr functions can therefore be transformed effortlessly, and their evaluation is flexible as needed.

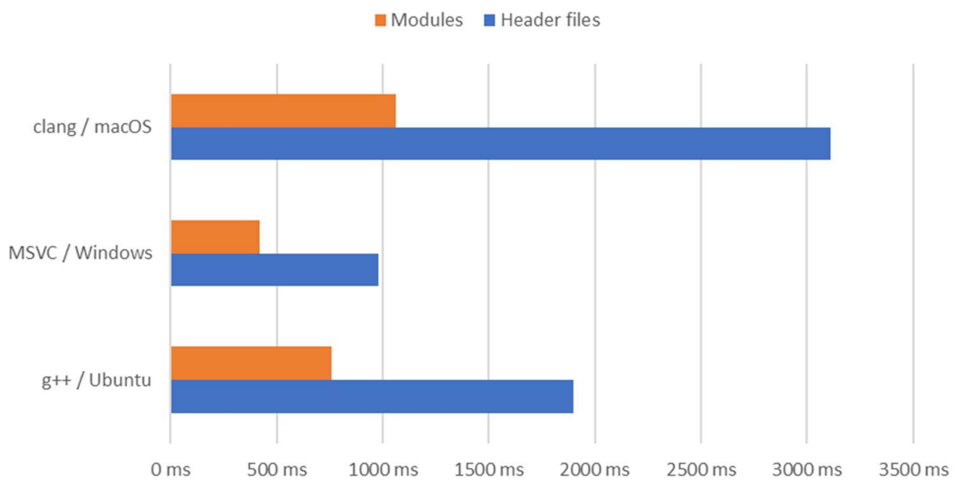
Modules or header files

Now let's take a look at the results from tested modules compared to header files. In case of using clang on macOS or g++ (Ubuntu), the compile size of whole programs for modules was in lower kilobytes, meanwhile the header files were in the lower units of megabytes.

The memory savings when compiling with g++ in a computing setup with Ubuntu was several percent. Huge difference was when using bits/stdc++ library, where the iostream was sufficient for modules. This resulted in up to six times of memory savings for the modules.

The chart in Figure 6 compares programs using only modules or header files as a translation unit.

Figure 6 Comparison of compilation speed



Compilation time in this case is probably one of the most remarkable things. Obtained values were statistically evaluated and added to the graph, using different configurations and operating systems. All the used compilers achieved excellent results, although each of them approaches modules slightly differently and the current library support is not the same for all of them.

5 Conclusion

In our work, we concentrated on the different techniques that are suitable for green cloud computing data centres. One of the easily affectable areas is the development of energy-efficient applications. The application efficiency depends on the used compiler and the used constructions within the source code. Our measurements experimentally confirmed that the energy requirements can be decreased by about tens of percent. The efficient application development seems to be a very promising building stone for green cloud computing data canters.

Acknowledgment

The authors would like to acknowledge the University of South Bohemia for the provision of the environment and resources used for the creation of this paper.

References

Bharany, S., Sharma, Osamah, I., S., Khalaf, O. I., Abdulsahib, G. M., Al Humaimeedy, A. S., Aldhyani, T. H. H., Maashi, M., Alkahtani, H. (2022). A Systematic Survey on Energy-Efficient Techniques in Sustainable Cloud Computing. *Sustainability*, 14(10), DOI: 10.3390/su14106256.

Lefevre, L., Orgerie, A. (2010). Designing and evaluating an energy efficient Cloud. *J Supercomput* 51, 352–373. DOI: 10.1007/s11227010-0414.

Nordman, B. and Berkeley, L. (2009). *Greener PCs for the Enterprise* no. August, 2009, DOI: 10.1109/MITP.2009.71.

Younge, J., von Laszewski, G., Wang, L., Lopez-Alarcon, S., Carthers, W. (2010). Efficient resource management for Cloud computing environments. *International Conference on Green Computing*, Chicago, IL, 357-364, DOI: 10.1109/GREENCOMP.2010.5598294.

Kim, J. M., Kim, M., Kong, J., Jang, H. B., Chung, S. W. (2011). Display Power Management That Detects User Intent. *Computer*. 44(10), 60–66. DOI: 10.1109/MC.2011.312

Lin, C. (2012). A Novel Green Cloud Computing Framework for Improving System Efficiency. *Physics Procedia* 24, 2326–2333. DOI: 10.1016/j.phpro.2012.02.345,

Ketankumar, D. C., Verma, G., Chandrasekaran, K. (2015). A Green Mechanism Design Approach to Automate Resource Procurement in Cloud. *Procedia Computer Science*, 54, 108–117. DOI: 10.1016/j.procs.2015.06.013.

Khanna, R., Zuhayri, F., Nachimuthu, M., Le, C, Kumar, M. J. (2011) Unified extensible firmware interface: An innovative approach to DRAM power control. *2011 International Conference on Energy Aware Computing (ICEAC)*. IEEE. 1–6. DOI: 10.1109/ICEAC.2011.6136703.

Esmaeilzadeh, H., Cao, T., Xi, Y., Blackburn, S. M., McKinley, K. S. (2011). Looking back on the language and hardware revolutions: measured power, performance, and scaling. *ACM SIGARCH Computer Architecture News*. vol. 39. ACM. 319–332. DOI: 10.1145/1961295.1950402.

Chen, Y., Chen, T., Xu, Z., Sun, N., Temam, O. (2016). DianNao family: energy-efficient hardware accelerators for machine-learning. *Communications of the ACM*. 59(11), 105-112. DOI: 10.1145/2996864.

Mashayekhy, L., Nejad, M. M., Grosu, D. (2015). Physical Machine Resource Management in Clouds: A Mechanism Design Approach. *IEEE Transactions on Cloud Computing*, 3(3), 247–260. DOI: 10.1109/tcc.2014.2369419.

Singh, K., Ku, M.-L. (2015). Toward Green Power Allocation in Relay-Assisted Multiuser Networks: A Pricing-Based Approach. *IEEE Transactions on Wireless Communications*. 14(5), 2470–2486. DOI: 10.1109/twc.2014.2387165.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., Saraiva, J. (2021). Ranking programming languages by energy efficiency. *Science of Computer Programming*. 205, 102609. DOI: 10.1016/j.scico.2021.102609.

Calandra, A. (2022). *Modern C++ features*. https://github.com/AnthonyCalandra/modern-cpp-features Microsoft documents. C++ Standard Library header files, https://learn.microsoft.com/en-us/cpp/standard-library/chrono?view=msvc-170 cppreference.com. Modules (2022), https://en.cppreference.com/w/cpp/language/modules

Lischner, R. (2020) *Exploring C++20: The Programmer's Introduction to C++*. 3rd ed. Berkeley: Apress,

Stroustrup, B. (2020). *Thriving in a Crowded and Changing World: C++ 2006–2020*. https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf

Fertig, A. (2021). *C++20: A neat trick with consteval*. https://andreasfertig.blog/2021/07/cpp20-a-neat-trick-with-consteval

Stroustrup, B., Sutter, H. (2022) *C++ Core Guidelines*. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

Microsoft documents (2022). Overview of modules in C++, https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170

Stroustrup, B. (2018). *A tour of C++. Second edition.* Boston: Addison-Wesley. C++ In-Depth series.

ModernesCpp.com (2020). *C++20: The Advantages of Modules*. https://www.modernescpp.com/index.php/cpp20-modules.