České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Czech Technical University in Prague
Faculty of Nuclear Science and Physical Engineering

Ing. Miroslav Virius, CSc.

Fyzikální experiment COMPASS z hlediska informačních technologií

Physical Experiment COMPASS in terms of the Information Technology

Summary

COMPASS is an international fixed target experiment in CERN at the SPS accelerator. Its goal is to study the hadron structure and the hadron spectroscopy by means of the high energy beams of hadrons and muons.
Heavy use of information technologies is indispensable in any high energy physics experiment. It is applied not only during late stage of the data processing, but also in the data acquisition process, control and checking of the devices, producing test data for the purpose of analytical software debugging and testing, etc. The object oriented approach on one hand and the efficiency of the final code on the other hand were emphasized during the software design and development process. The C++ programming language, used for the software development, was new for many programmers in the time when the software development started. (Up to middle 90s, Fortran was used in CERN as the main programming language.) Thus, not only the object oriented programming theory, but also the C++ features leading to inefficiencies, and innovative optimization methods, such as template metaprogramming, were investigated to overcome these problems.
General aspects of the COMPASS experiment, as well as author's contribution will be discussed in the presentation.

Souhrn

COMPASS je mezinárodní fyzikální experiment s pevným terčem v CERN na urychlovači SPS. Jeho cílem je studium struktury hadronů a hadronové spektroskopie za pomoci vysokoenergetických svazků hadronů a mionů.
Jakýkoli experiment v oboru fyziky vysokých energií je nemyslitelný bez rozsáhlého nasazení informačních technologií. Jedná se nejen o zpracování naměřených dat, ale i o sběr dat a jejich předzpracování, ovládání a kontrolu přístrojů, získávání testovacích dat pro ladění a testování analytického softwaru před započetím sběru dat atd. Při návrhu a vývoji softwaru byl kladen důraz na jedné straně na objektově orientovaný přístup a na straně druhé na efektivitu vytvořeného kódu. Situaci komplikovala skutečnost, že v době, kdy tvorba softwaru pro COMPASS začínala, byl pro většiny programátorů jazyk C++, použitý při vývoji softwaru, nový (do té doby se v CERN používal jako hlavní programovací jazyk Fortran). Proto jsme se věnovali nejen teorii objektově orientovaného programování, ale i vlastnostem jazyka C++, které mohou působit ztrátu efektivity, a netradičním optimalizačním technikám, jako je šablonové metaprogramování.
V prezentaci budou diskutovány jak obecné aspekty experimentu COMPASS, tak i přínos autora.

Klíčová slova: COMPASS, sběr dat, ROOT, COOOL, objektově orientované programování, optimalizace, šablonové metaprogramování, generické konstrukce

Keywords: COMPASS, DAQ, ROOT, COOOL, Object Oriented Programming, Optimization, Template Metaprogramming, Genericity

Table of Contents

1. Introduction: The COMPASS Experiment

COMPASS is an international high-energy physics fixed target experiment that runs at the SPS (Super Proton Synchrotron) accelerator at CERN in Geneva, Switzerland. The main purpose of this experiment is to study the hadron structure and hadron spectroscopy with high intensity muon and hadron beams.

The code of this experiment at CERN is NA-58.

1.1 Brief History

This experiment was proposed in 1995, approved conditionally at CERN in February 1997 and finally approved in September 1998. Technical run started in 2000, physics run, as well as the data acquisition of this experiment started in 2002. More than 240 scientists from 28 universities and research institutes in 10 countries take part in this experiment now, including the Joint Czech Group.

The proposed setup and the initial setup of the COMPASS experiment are shown in Fig. 1.

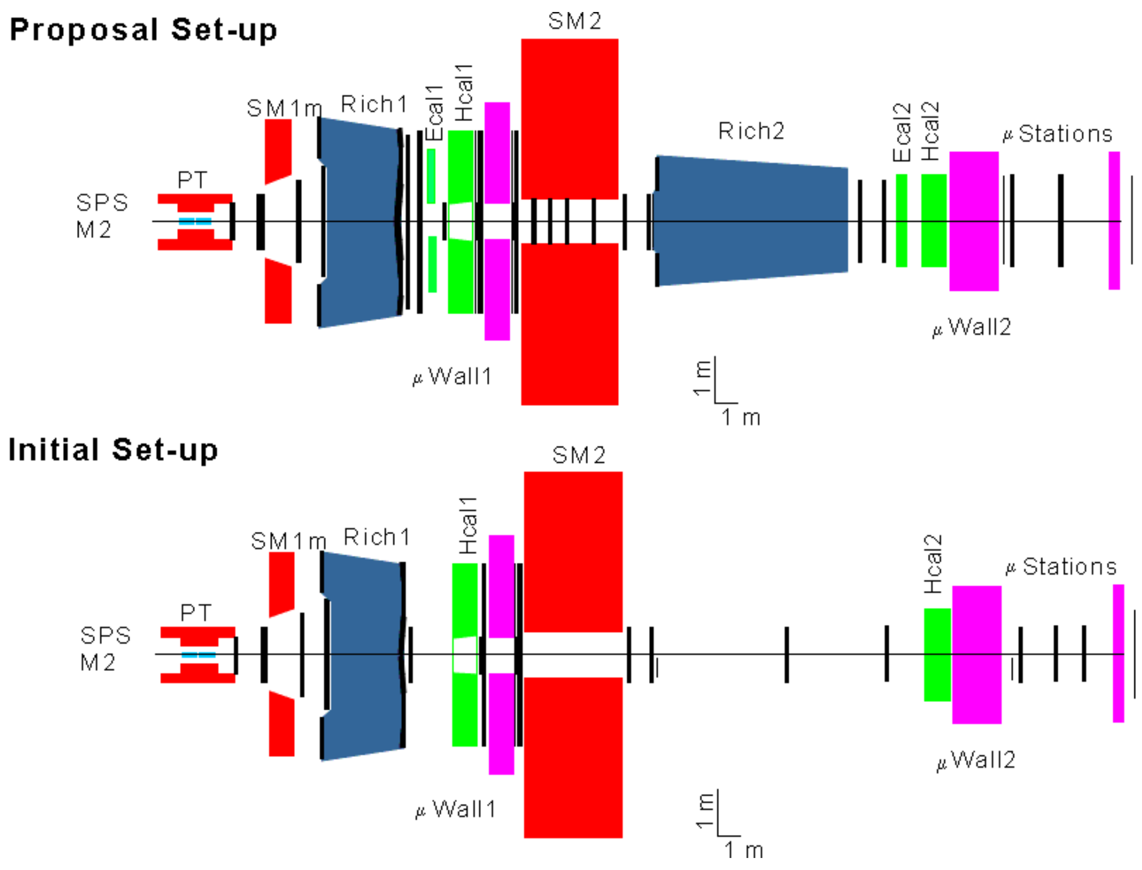

Fig. 1 Proposed and initial setup of the COMPASS experiment [2]

In this figure, the particle beam from the accelerator comes from the left side and hits the polarized target (PT), that consists of the ^{6}LiD or NH_3 (the first behaves like polarized deuterons, the second like polarized protons). This target is cooled to the temperature about 50 mK and is embedded into the intense magnetic field – this allows to achieve and hold the polarization of the nuclei (deuterons or protons) in the target.

Behind the polarized target follows the line of detectors. Note that the initial setup did not contain all the proposed detectors – these devices were constructed and installed later and this process continues up to now.

1.2 The COMPASS aim

COMPASS was established as a union of two different initially proposed experiments, so it runs in two basic modes – with muon beam and with hadron beam [3].

The aim of the COMPASS experiment at CERN is to study in detail how nucleons and other hadrons are made up from quarks and gluons. At hard scales, Quantum Chromodynamics (QCD) is well established and the agreement of experiment and theory is very good. However, in the nonperturbative regime, despite the wealth of data collected in the previous decades in laboratories around the world, a fundamental understanding of hadronic structure is still missing.

The COMPASS Experiment uses muon and hadron beams of very high intensity from the SPS accelerator at CERN. It investigates the spin structure of nucleons a hadron spectroscopy. Outgoing particles (one or more) are detected in coincidence with incoming muons or hadrons.

Large polarized target embedded in a superconducting solenoid is used for the measurement with muon beam. Outgoing particles are detected by large angle spectrometer with large momentum range. The apparatus consists of tracking detectors of different types corresponding to the expected amount of collisions necessary for the requested spatial and angular resolution. For the particle identification, the RICH detector (Ring Imaging Cherenkov) is used, as well as hadron and electromagnetic calorimeters.

One basic feature of the COMPASS experiment is the detection of large statistical samples of charmed particles. The gluon polarization (ΔG) measurement is based on the measurement of the open charm cross section asymmetry in deep inelastic scattering of polarized muons on polarized nucleons. There are predictions of the value of this quantity based on some numeric calculations and on the QCD models.

The hadron beams enable us to study semileptonic decays of the charmed and doubly charmed baryons.

Both these measurements enable us to solve the basic problems of the structure of the hadrons and to examine the computations from the effective Heavy Quark Effective Theory (HQET).

The setup of the experiment allows to perform the basic physical measurements with very high precision and sensitivity. This permits to investigate the exotic states, that are predicted by the QCD, but were not yet observed.

The setup of the experiment allows to investigate both the longitudinal and the transverse spin distribution functions. The aims of the experiment may be summarized as follows:

Muon beam experiments

- Measurement of $\Delta G/G$,
- Measurement of g_1,
- Transverse spin effects,
- Flavor decomposition of spin distribution functions,
- Vector meson production,
- Spin transfer in Λ-hyperon production.

Hadron beam experiments

- Pion and kaon polarizability,
- Diffractive production of exotic states,
- Glueball search,
- Light meson spectroscopy,
- Double charmed baryon production.

1.3 Information Technologies in COMPASS

The main areas, where information technologies in COMPASS apply, are:

- Data acquisition (DAQ),
- Control of several parts of the apparatus,
- Slow control,
- Monte Carlo simulation,
- Data management
- Data analysis.

In this text, we will shortly discuss the DAQ and slow control. The main part will be devoted to special problems – mainly the optimization – of the software for the data analysis.

2. COMPASS Data Processing System

2.1 COMPASS Data Acquisition System

The COMPASS Data Acquisition system is derived from the DAQ system of the CERN LHC ALICE experiment.

2.1.1 Requirements and conditions

The beam that hits the polarized target consists of the so called “spills” of particles. One spill is random flow of about 2 . 10^8 particles, which are approximately uniformly distributed in time interval about 6 seconds long. It is followed by a pause about 12 seconds long.

Data from the detectors are collected by about 200.000 channels. The total data amount produced by one event, i.e. by the pass of one particle through the apparatus, is 40 – 50 KB. The exact value depends on the noise of the apparatus and on the settings of some triggers.

The total data amount for one spill is from 900 MB to 18 GB.

The data must be collected, preprocessed and saved; this is an on-line process. Next, the data are processed – analyzed; this is done off-line.

2.1.2 DAQ Architecture

The DAQ architecture consists of several layers. The first layer is the front end, i.e. the electronic equipment of the detectors. The front end converts analogue data to digital form, and gives it the format expected by the following layers. Beside this, the front end delays the data according to the distance from the polarized target to the given detector, so that all the data form one particle have the same time stamp. This enables to scan the events in this “time windows”.

The next layer consists of the so called CATCH computers. (For GEMs and silicon detectors, there is a specialized version GeSiCa.) These computers are interconnected by the VME high speed bus. The CATCH computers check the formal consistence of the data and pass them to the next layer.

Serial optical link to the next layer is based on the S-link standard developed at CERN. This is serial FIFO and connects the CATCH computers to Read Out Buffer computers (ROBs). ROBs serve as buffers for the collected data. Data wait in this layer to be taken by the Event Builder layer (EVB).

The EVBs are interconnected with the previous layer by the gigabit Ethernet network available at CERN. EVBs eliminate unnecessary data and create files ready for further processing from the correct data. In these files, the data are composed into events – i.e. in the form describing the pass of single particles.

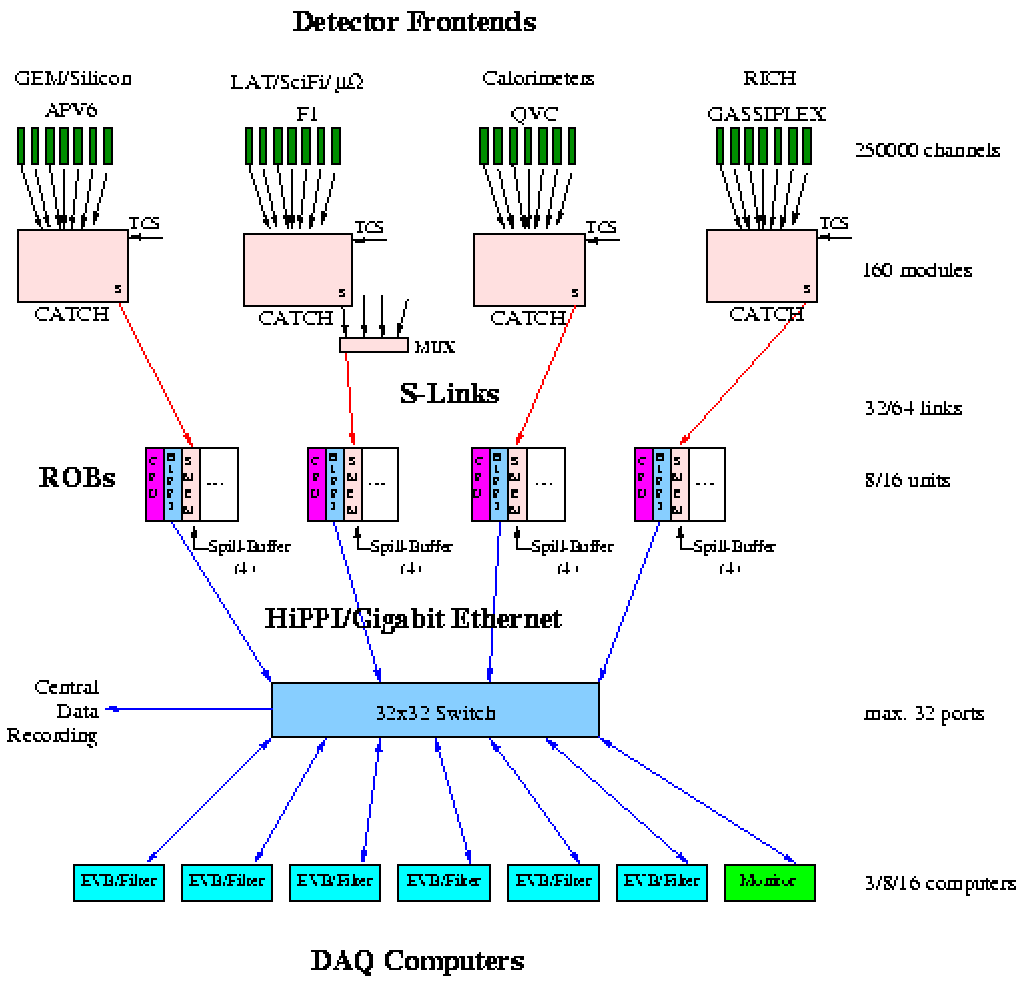

Fig. 2 DAQ of the COMPASS experiment. [2]

Data are stored in EVBs at least 24 hours. After that, they are transferred to the Central Data Recording.

Fig. 2 shows the schema of the DAQ system of the COMPASS experiment.

2.3 Data Migration to CDR

Central Data Recording is a tool for the transfer of the raw physical data from the on-line running DAQ system to the permanent data store. CDR exploits the CERN network infrastructure. It is based on the CASTOR system, that was developed at CERN.

Castor runs now on 4 dedicated Linux file servers with RAID; these disks are organized as an array with 10 file systems.

A specialized computer inspects the EVBs twice per hour, selects data older than 24 hours and transfers them to CDR. Data are stored on magnetic tapes. At the same time, metadata describing the stored files are generated. The metadata are stored in the Oracle database system.

2.4 Data organization

Data are stored in the CASTOR system. They are divided into the so called *chunks*. A set of the chunks containing data from all the detectors is called *run*. A run together with the metadata and data in the log-book (log of the measurements) contains all the necessary information for the determination of the conditions (and problems) that occurred during the data taking for that run.

Average size of one chunk is 0.82 GB; average run contains 200 chunks.

2.5 COOOL

The COOOL (COMPASS Object Oriented On-Line) program system is independent on the DAQ. The COOOL is a system of programs, that takes data from the CASTOR, or takes them on-line and preprocesses them: It finds data describing trajectories of single particles, rings indicating some particles in data from the RICH detectors etc.

COOOL contains the tools for the data visualization, too.

The COOOL is based on the ROOT system, which is an object oriented programming environment for data analysis written in C++. Some parts of the COOOL system serve for the graphic output used by the shift crew to check the examples of the measured data. This allows to check, whether all the apparatus works in preset bounds.

Output of the COOOL system may be stored to be used later by the ROOT system.

2.6 Data Processing

We need to extract relevant data from the raw data stored on the CASTOR system; it follows from the size of the data, that the time necessary for these computations is a very important criterion. Special analytical program packages, named CORAL (*CORe AnaLysis*) and PHAST (*PHysics Analysis Software Tool*) are used for this purpose.

Both the program packages need many subsidiary software tools; it is necessary to preset the environment for the programs, too. This is done by means of the so called *option files*. The options in these files may be considered as a kind of metadata controlling the behavior of the analytical software.

The processing runs on the cluster at CERN. It follows that it is necessary to balance the load of single computers as uniformly as possible. Nevertheless, it is necessary to be aware which data were already processed and which are still waiting to be processed.

The cluster provides the mechanism for submitting a *job* for processing. The user has no means to access single nodes of the cluster, it is all solved by means of the cluster software.

We implemented our own mechanism (see Fig. 3) for the control of the processing.

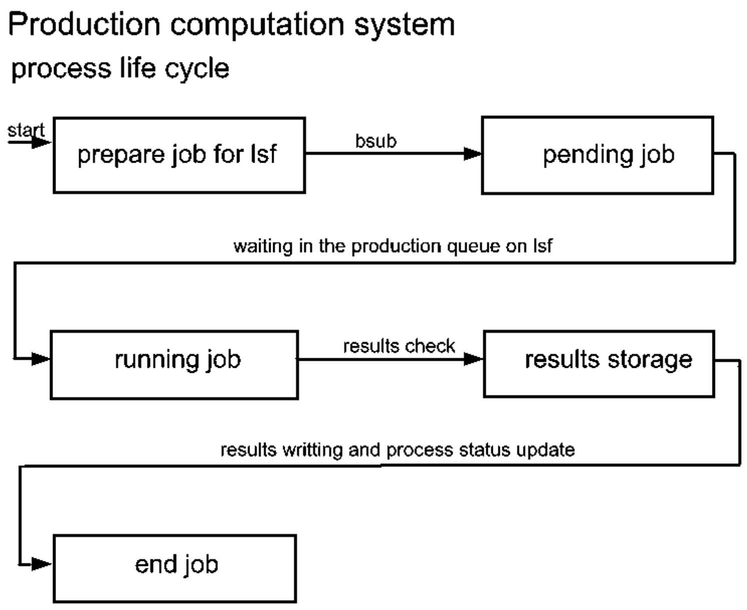
Production computation system
process life cycle
start
prepare job for lsf
bsub
pending job
waiting in the production queue on lsf
running job
results check
results storage
results writting and process status update
end job

Fig. 3 Diagram of the data processing

This system has the following properties:

- The system can prepare one file for the processing.
- The system can set the environment of the cluster so that it corresponds the needs of the analytic software tools.
- The system can communicate with the cluster.
- The system can submit a job for the processing.
- The system can identify the current state of the job.
- The system is aware of the current state of all the computations
- The system can show the summary of the current sate of the system it the form of the tables and graphs.
- The system can properly respond to the errors of the system (data storage is not available, the processing is interrupted for reasons of the breakdown of the computing node, there is an error in the analytical software, etc.)
- The system can process partial results of single jobs so that they are stored in the form suitable for the physicist that will evaluate them.

The system detects non standard behavior of the job, and in the case of an error during the processing, it tries to recover (it prepares the job once again and resubmits it) or it informs the administrator by an e-mail.

3. Object Oriented Computing

The IT division of CERN decided to stop supporting the Fortran programming language, which had been used as the main programming language for the scientific computation, in the middle of 1990s. It was replaced by the C++ language. Special program libraries, such as the GEANT simulation library, had to be translated to the C++ language.

Some problems emerged:

- Object oriented computing, which is the most prevailing programming paradigm now, was new in the 90's for many programmers.
- The C++ programming language contains some constructs, that appear to be relatively ineffective under some circumstances. It was necessary to analyze the language and to find those ineffective constructs, that may appear in the analytical software.
- In cross check computation, some new optimization techniques were tested. This is the case of the so called template metaprogrammig.
- The differences of the C and C++ languages may be confusing in some context. Analysis of the relation of the C++ standard (issued in 1998) and the new C standard (issued in 1999) was necessary.

3.1 Basics

Object oriented programming (OOP) is a methodology for the analysis of the requirements imposed on the software to be developed and for the development process. It is prevailing methodology now.

Object oriented program consists of the program constructs called *objects*, that cooperate by sending messages to each other, and of course they respond to these messages. The OOP is based on four basic concepts in the simplest form – the *class*, the *encapsulation*, the *inheritance* and the *polymorphism*.

The *class* is the same as an *object data type*. It is a data type describing some class of concepts from the problem domain – it serves as a program model of this concept. Of course this model reflects only the aspects of the concept that are important for the solution of the problem.

An *instance* of the class is an *object* in object oriented program.

The class is given by

- the data representation (data components of the separate instances and data of the class, i.e. data shared by all the instances), and
- the operations that may be performed with single instances or with the class as a whole – the *methods*.

The *encapsulation* is the hiding of the implementation and the separation of the interface and the implementation. No class should publish (present to other parts of program) more parts of the implementation than necessary for the use of it in the program.

Note that the encapsulation concept is not new in OOP; this principle was first formulated in the modular programming in early 60s. In the context of the modular programming, the encapsulation is applied to the module; in the context of the OOP, it is applied to the class.

The object oriented programming languages contain many tools that support – or even enforce – the encapsulation on the syntactic level. This enables to transform the encapsulation violations – i.e. the possible errors of the program design – into syntactic errors, that may be found by the compiler.

The *inheritance* is a tool that allows us to derive a new object type form an existing one (the so called *base* type). The derived type inherits all the properties of the base type – i.e. it has all the data components and all the methods. The programmer can add new methods and data components; he may override existing method (define new implementation of it). He may not remove any method or data component.

The derived class is always a *subclass* (specialization) of the base class. It describes more special concept than the base class.

The concept of *polymorphism* represents the possibility to deal with different instances of different classes in the same way. It means that we can send the same message to a group of instances of different classes and all the instances will accept it and any of them will process it in its own way.

In the most common programming languages, such as C++, the polymorphism is achieved through the inheritance. Newer programming languages, as Java or C#, use the implementation of the interfaces as another way to achieve the polymorphism.

Object oriented analysis uses the Unified Modeling Language (UML). This is the set of rules for the creation of diagrams describing different aspects of the solved problem and designed program. Here are some most common UML diagrams:

- The *class diagram* describes the classes in the solved problem and the relations (inheritance, composition, association etc.) of these classes.
- The *object diagram* describes the objects (i.e. the instances of the classes) in the solved problem and their relations.

- The *activity diagram* describes the activities of the objects in the solved problem. It may be used for the description of the algorithms.
- The *use case diagram* describes ways the described system is used.
- The *state diagram* describes the states of the system or of its parts and the possible transitions between these states.
- The *deployment diagram* describes the deployment of the application, i.e. the placement of different components of the application on different computers of the target system.

3.2 Object Oriented computing in the COMPASS Experiment

The Joint Czech Group in the COMPASS experiment participated in the preparation of this experiment since the beginning, even though at the start it was under the flag of JINR (Dubna). One very important part of the preparation was the development of the analytical software. In this stage, it was necessary to decide, what parts of the software will be developed from scratch, what third party software will be customized (this were usually programs used by previous or running experiments – this is the case e.g. of the DAQ system), etc.

One the first task was the program model of the prepared experiment COMPASS based on the Monte Carlo Method. This model served

- to examine some alternatives in the experiment setup, and
- to get test data for the debugging of the analytical software that was developed.

The program model of the apparatus consisted of the models of its parts. Some of its basic components were e.g.

- the *particle* class, describing an abstract particle,
- the *target* class, describing the polarized target,
- the *detector* class, describing an abstract detector.

The abstract classes, like detector and particle, served as base classes for the so called instance classes describing concrete detectors and particles.

Similar models were used in other software pieces developed for the control of the apparatus etc.

I participated on the design of this model and used its basic parts in [3]. Of course, the [3] does not contain the whole model, but only some selected parts of it.

4. Migration to C++ – Efficiency Problems

Owing to the amount of the data, that should be processed, the efficiency of the analytical software is crucial. The migration of the programmers from Fortran to C++ brought some unexpected problems in this area. The C++ programming language contains some constructs, where “no code is written, but the compiler generates some". The first analysis of this language was aimed to this "secret code" and to the ways how to avoid it. The result was published in [4].

Three basic levels of program optimization are usually distinguished:

- The algorithmic level. This is the topmost level optimization and is of course the most important. This is done solely by a programmer; a good example is the choice between the bubblesort algorithm, which has the $O(n^2)$ complexity, and the quicksort algorithm with complexity $O(n.\log_2 n)$.
- The second level is performed by the compiler in cooperation with the programmer: The programmer indicates to the compiler, what optimization it could do. A good example of this level of optimization is the *inline* modifier in the function declaration or the *register* modifier in the local variable declaration. The effect of the optimization on this level is usually less than the effect of the algorithmic optimization.
- The third level is performed solely by the compiler, sometimes even in the case when the programmer disables all the optimizations. This optimization consists of the common subexpression elimination or the substitution of one instruction by another, which produces the same result, but faster. The results of this level optimization is usually negligible in comparison with the other two levels.

4.1 Sources of C++ inefficiency

Optimizations based on the result of the C++ inefficiencies analysis is usually on the second level, even though there are examples, when it may change the order of the complexity of the algorithm.

The following main sources of the inefficiencies in the regular C++ code were found:

- automatic conversions using the conversion constructor,
- automatic conversions using the conversion operator defined by the programmer,
- automatic constructor and destructor call,
- superfluous use of named local variables.

In the first three points, the programmer will not notice the potential inefficiency, because he or she does not write any code; the code is generated by the compiler.

The automatic conversions in first two points may lead to the creation of temporary variables of object types. This produces additional constructor and destructor calls; under some circumstances, it may cause even the use of an algorithm with worse complexity.

To the last point: The compiler can optimize out *anonymous* temporary variables. It is allowed to replace the creation of the temporary variable containing the return value of a function and the process of copying it to the destination address by direct creation of the result in the destination address (this is the so called *return value optimization*). On the other hand, the compiler is not allowed to remove ("optimize out") any named local variable during the optimization process.

4.2 What is not source of inefficiency

The programmers often believe that OOP as a whole is inefficient. The same delusion about the operator overloading is quite common.

Analysis of the disassembled code of OOP programs in C++ shows that this is not the case.

An object is usually stored in the computer memory in the same way as the C structure and method call is a function call with one parameter more (pointer to the object, on which the method is called). Thus the use of objects in C++ has in many cases the same complexity as the use of the structures in analogous structured (not object oriented) program.

The possible source of the inefficiency on OOP is the virtual method call and the use of virtual inheritance.

The algorithm of the virtual method call in C++ seems to be quite complex, but analysis of the disassembled code shows, that only one additional instruction per method call is necessary (in comparison with the call of nonvirtual methods).

If the virtual inheritance is used, some data members of the instance must be accessed by means of hidden pointers stored in the instance. This brings some overhead, but the access of the virtually inherited data members is a constant time algorithm.

Note that virtual inheritance is feature that is seldom used in the programs and in physical computations is not used at all.

The operator overloading is a syntactic feature of the language, that affects the compiler complexity, but it does not affect the complexity of the produced code. The use of an overloaded operator in the source code is equivalent to the function call and this is the way it is compiled.

To be more precise: Overloaded operators are in C++ code declared as functions or as methods of object types. Given two instances, `a` and `b`, of object type `T`, and an overloaded operator +, that accepts operands of type `T`, the expression `a + b` is compiled as a function or method call in the form `a.operator+(b)` or `operator+(a, b)`, depending on the declaration of this operator.

It follows that the use of the overloaded operators leads to the programs of the same complexity as the use of the functions or the methods computing the same result.

Of course, the use of automatic conversions, superfluous local variables etc., as discussed in the previous section, may lead to inefficiency in the overloaded operators as well as in usual functions and methods – but this is not caused by the operator overloading.

5. Template metaprogramming in C++ as an optimization tool in physics computation

Template metaprogramming is an advanced tool for metacomputation, i.e. for the computation in the time of the compilation. It is based on the C++ templates and may be used for some kind of control of the compiler and for optimizations like loop unwinding. The use of the template metaprogramming was considered as an optimization tool in the design of the cross check programs for the COMPASS experiment.

5.1 Basics

The C++ *template* is a tool for the production of an infinite set of classes, methods or functions in the program. Members of that set differ only in some data types or integer constants. These data types and constants serve as template parameters. Note that the term *generic types* or *generic functions* (or simply *generics*) is often used instead of the term *templates* in the C++ language.

The original purpose of the C++ templates was to simplify the programming of the co called containers (or collections) – data types like lists, queues, hash tables etc., serving as a store for some amount of data, and common algorithms, such as computing the maximum or minimum of two numbers, sorting a container or applying a transformation to all the data pieces stored in an array.

The C++ programming language allows the programmer to define the following constructs side by side in one program:

- The *primary template*. This is a common template of the class that is valid for all the data types and for all the values of the constants that are used as template parameters.
- The *partial specialization*. This is another template with the same name that is valid for a selected group of data types or for selected values of the constants and selected (nonempty) group of data types passed as template parameters.
- The *full (explicit) specialization*. It is the template definition for a given data type or types and/or for selected values of the constants passed as template parameters.

With the template mechanism of the C++ programming language, it is easy to map existing data types to another data types, as well as the integer constants to data types.

5.2 Computational completeness

It is easy to prove that the templates can be considered as a computationally complete tool (in the domain of integer numbers) for the generation of the data types.

Template mechanism enables us to map data types and integer constants to data types of special kind (and to retrieve the mapped type or value).

It gives us the way implement the decisions in the algorithm of the metaprogram, i.e. to declare different data types depending on some condition, that is evaluated during the compile time. This is based on the partial specialization mechanism.

There is a way to program the loops in metaprograms, too. This is done in a way very similar to the implementation of the loops in functionally oriented programming languages, e.g. in the Lisp programming language. There are no explicit loops in the programming languages of this kind, the loops are replaced by the recursion:

- The class template with an integer parameter may refer to an instance of the same template with another argument. This forces the compiler to create recursively a sequence of different instances of the same template.
- The recursion may be stopped using partial or explicit specialization for the final value of the template parameter.

It is easy to write metaprograms that compute e.g. the factorial of a given number in the compile time; an example may be found in [5].

Note that all the above mentioned constructs do not need to reserve any memory, thus they do not impose any overhead to the resulting program.

5.3 Loop unwinding

One possible use of the metaprograms is in program optimization, mainly in the loop unwinding.

The loops appear very often it the physical computations. Usual optimizations in the compilers suppose that the number of the iterations in the loops are very large; this is contra productive in the case of the loops with a very small number of iterations that are embedded in the loops with very large number of the iterations. This is e.g. the case of the dot product of two vectors with 2 or 3 components in a many times repeated loop.

It is easy to see that the best optimization of such a computation is to unwind the inner loop (computing e.g. the dot product), i.e. to write a linear code instead of the loop multiplying the components and summarizing the partial products.

On the other hand, if we do not know the number if iterations of the inner loop in the design time, such an optimization requires to write many times the code that is almost the same with only slight changes. This violates one the basic programming rule, which is “do not repeat yourself”. That would probably cause some problems in the debugging and maintaining the final code in larger programs.

If the number of the iterations in the loop can be ensured in the compile time, i.e. if it can be represented as a constant in the program, the template metaprogramming enables us to write a metacode that generates the linear code in the final program and thus it may produce more effective code that the usual code based on the loops (and without the repetitions in the source code).

So far the theory. Of course, the applicability of the metaprogramming based optimizations is affected among other by the quality of the compiler, thus the testing was necessary. The `N` times repeated computation of the dot product of two vectors with 3 components were used as a test example with `N == 10`^{9}. The time necessary for the computation was measured by the `_ftime()` function. Note that this function is not the part of the C++ standard library, but it is a usual extension of the C run time library.

We have compared four compilers, that were popular with many programmers – the Borland C++BuilderX (BCX), the Microsoft Visual C++ 2003 (MSVC), the Intel 7.1 and the GNU g++. Mainly different versions of the Borland and Microsoft compilers were often used for the program development and debugging; on the other hand, the production compiler is GNU g++. The following table shows the results; average time of 10 program runs is given in seconds, all compilers were used with full optimization setting.

	BCX	**GNU**	**Intel**	**MSVC**
Loop	72.3	83.5	42.2	15.5
Metaprogram	10.8	52.3	2050	5.3
Linear code	3.0	4.1	2049	5.3

Absolute values of the time depends on the quality of the computer used and is not relevant in this comparison, even though it may be taken as a kind of benchmark for the compilers. The interesting result is the relative comparison of the times for different compilers and for different forms of the source code, which is in the next table; percentage share of the time for unwound loop is given.

	BCX	**GNU**	**Intel**	**MSVC**
Loop	100 %	100 %	100 %	100 %
Metaprogram	14.9 %	62.6 %	--	34.2 %
Linear code	4.1 %	4.9 %	--	34.2 %

As you can see, the time savings due to the metaprogram based optimization varies – depending on the compiler – in the approximate range of 35–75 % with respect to the code containing the unwound loop.

Note that the Intel compiler failed at all.

6. Generic programming – the implementation

Template metaprogramming is a tool specific for the C++ programming language. Nevertheless, generic constructs are available in many modern programming languages, as in Java or C#. Programmers involved in the COMPASS experiment often use these programming languages to design and test the algorithms, even though the final implementation is done in C++.

Thus, the necessity of the analysis of implementation of generics in different programming languages arises.

6.1 What is generics

Let us start with the formal explanation of the concept of generics; up to now we have discussed the generics only in the context of the C++ programming language.

Generic types, also known as parameterized types, are object data types, whose declaration contains the so called formal types – type names without specified exact meaning in the moment of the declaration – where data type names are expected. When the generic types are used, all the formal types are replaced by particular types passed to the generic type by the programmer.

Similarly, the generic method is a method, whose declaration contains formal types instead of some data types – usually instead of the types of the formal parameters of the method or instead of the return type. When the generic method is used, the formal types are replaced by particular types. These particular types may be passed by the programmer, but sometimes they may be deduced by the compiler from the context.

6.2 Main goals of the generics

The main goals of the generics in common programming languages, as C++, Java or C#, are [6]:

- To provide means to express the algorithms as independent on the data structures as possible.
- To provide means to create data structures as independent on the algorithms for the manipulation with it as possible.
- To provide means to implement algorithms in as generally as possible without any loss of the efficiency.
- If the general form of the algorithm is inefficient, unsuitable or unusable in some special cases, the goal is to provide a special implementation for these special cases.
- If the general form of the data structure is inefficient, unsuitable or unusable in some special cases, the goal is to provide a special implementation for these special cases.
- If there is a couple of equivalent algorithms for the solution of the same problem, the goal is to provide a way to implement all these algorithms and to let the programmer to choose one according to some other criteria.

Implementation of generics

There are at least three main approaches to the implementation of the generics.

- Generic constructs exist only in the source code and serve as a template for the compiler for the creation the instances of the generic type or generic method. Only the instances created this way exist in the executable code. Different instances of one template represent different data types or different methods.
- Generic constructs exist only in the source code and announce the compiler, that it should use stricter type checking. This mechanism produces only one data type or one method based on one generic construct; unlike in the previous case, there are no instances, even though the program may behave as if there were ones.
- Generic constructs in the source code are compiled to generic constructs in the executable code. The executable code produces instances in the run time.

The first approach is used e.g. in the C++ compilers. The C++ templates behave in fact like macros, even though they are evaluated by the compiler, not by the preprocessor. This approach imposes no overhead in runtime; its drawback is, that the compiler needs the source code of the generic construct for the compilation of its application.

The second approach is applied e.g. in the Java programming language. The basic point of this approach, that the formal types are treated as the most general type in this language at all – the java.lang.Object type. All the occurrences of the generic type or method in the program refer to the same code in compiled program; the compiler adds some run-time type checking.

This approach produces less efficient code than the first approach (this is due to the run-time type checking). On the other hand, the compiler does not need the source code when it compiles some code referring to the generic constructs.

The third approach is partially applied e.g. in the .NET Framework (e.g. in the C# language), where the compiler produces executable code containing the generic construct; for different value types, different instances are produced in the run time. For the reference types, only one instance (common for all these types) is produced. (Thus, the second approach is applied to the reference types in .NET.)

The compiler does not need the source code of the generic construct in the case of the third approach; on the other hand, there is runtime overhead caused by the creation of the instances and by run-time type checks.

References

1. The COMPASS Collaboration: *Propsal. Common Muon and Proton Apparatus for Strucrure and Spectroscopy*. CERN/SPSLC 96-14
2. http://wwwcompass.cern.ch/compass/detector/welcome.html
3. COMPASS Collaboration (P. Abbon, …, M. Virius et al.): *The Compass Experiment at CERN*. NIMA 577 (2007), str. 455 – 518.
4. M. Virius: *Object Oriented Computing*. In: J.E. Gentle, W. Hrdle, Y. Mori (eds.): Handbook of Computational Statistics, p. 403–434. Springer Verlag, Berlin 2004
5. M. Virius: *Migration to C++. Efficiency problems*. In: Proceedings of the International Workshop on Symmetry and Spin, Praha 5. – 12. 9. 1999, p. 371–374.
6. M. Virius: *Template Metaprogramming in C++ as an Optimization Tool in Physics Computation*. In: Proceedings of the Advanced Studies Institute Symmetry and Spin, Praha 19. – 26. 7. 2006, p. F353 – F360.
7. M. Virius: *Generické programování – Cíle a možnosti implementace*. In: Torba softwaru 2005. VŠB Ostrava, ISBN 80-248-1082-4, str. 38 – 45.

Curriculum Vitae

Ing. Miroslav Virius, CSc.

Born: 1953

Education:

1976	**Ing.:** Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering
1987	**CSc.** in Nuclear and Subnuclear Physics: Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering

Employment:

August 1, 1976	Employee of the Czech Technical University in Prague, Faculty of Nuclear Sciences and Physical Engineering, Dept. of Mathematics
May 1, 1981	Senior lecturer, Dept. of Mathematics, Faculty of Nuclear Sciences and Physical Engineering
1996	Member of the Dept. of Software Engineering in Economy, Faculty of Nuclear Sciences and Physical Engineering
2006 —	Deputy-head of the Dept. of Software Engineering in Economy

Research projects:

1977—1978	Collaboration with the Slovnaft company – solution of large systems containing linear and nonlinear equations
1978—1986	Member of the research team of project I-4-3/7-1 “Mathematical processing of the experiments with oriented radioactive nuclei” (part of the SPIN project in collaboration with the JINR, Dubna) – mathematical processing of the gamma spectra
1984—1990	Collaboration with SVÚM (Research institute of materials) – mathematical modeling of the neutron multiplier
1995—	Member of the preparatory group of the Czech participation in COMPASS
2002—	Member of the Joint Czech Group in the COMPASS Experiment (NA-58) at CERN, Genève, Switzerland)
2006—	Deputy-head of the Joint Czech Group in the COMPASS Experiment

2005—	Member of the Joint Czech Group in the PHENIX Experiment at Brookhaven National Laborarory, Upton, N.Y., USA
2005—	Member of the Institution Board of the PHENIX Experiment

University courses:

Monte Carlo Method
Introduction to Programming
The C and C++ Programming Languages
The Java Programming Language
Basic Algorithms
Programming for the .NET Framework

Object Oriented Programming (seminary on advanced programming techniques)

Additional notes

Author of a chapter in monograph published by Springer Verlag, invited by the editor.
Co-author of more than 40 papers by the COMPASS and PHENIX collaborations, published in impacted journals and international conference proceedings. Total number of citations exceeds 500 (including self-citations by other co-authors).
Research results published as contributions on international and local conferences.
Co-editor of the proceedings of international conferences (10).
Author or co-author of 19 books on programming; some of them are used as textbooks on Czech universities and secondary schools.
Author of 7 university textbooks for FNSPE; at least four of them are or were used outside FNSPE.
Author of 3 supplementary chapters for Czech translations of books on programming (included on the request of the publisher).

Author of about 700 popularization articles dealing different aspects of programming, programming languages, programming technologies, common errors in programming etc. in Czech IT-oriented journals.

Top citations

The following table shows the number of citations of selected papers based on the http://slac.stanford.edu/spires server by April 15, 2008. The "total" column displays the total number of citations in printed or electronically published articles according to the server, the "pure" column shows the number with self citations removed.

No.	Article	total	pure
1.	PHENIX Collaboration (A. Adare, … M. Virius et al.). *Energy Loss and Flow of Heavy Quarks in Au+Au Collisions at s(NN)**(1/2) = 200-GeV.* Phys.Rev.Lett.98:172301,2007	67	43
2.	PHENIX Collaboration (A. Adare, … M. Virius et al.). *J/psi Production vs Centrality, Transverse Momentum, and Rapidity in Au+Au Collisions at s(NN)**(1/2) = 200-GeV.* Phys.Rev.Lett.98:232301,2007	65	38
3.	COMPASS Collaboration (E.S. Ageev, … M. Virius et al.). *Measurement of the spin structure of the deuteron in the DIS region.* Phys.Rev.Lett.98: 232002,2007	50	30
4.	PHENIX Collaboration (A. Adare, … M. Virius et al.). *Scaling properties of azimuthal anisotropy in Au+Au and Cu+Cu collisions at s(NN) = 200-GeV.* Phys.Lett.B612:154-164,2005	46	20
5.	PHENIX Collaboration (A. Adare, … M. Virius et al.). *Measurement of high-p(T) single electrons from heavy-flavor decays in p+p collisions at s**(1/2) = 200-GeV.* Phys.Rev.Lett.97:252002,2006	45	25
6.	COMPASS Collaboration (E.S. Ageev, … M. Virius et al.). *Gluon polarization in the nucleon from quasi-real photoproduction of high-p(T) hadron pairs.* Phys.Lett.B633:25-32,2006	42	25

Total number of citations of these 6 articles is 315; number of pure citations of these articles is 181.