EPiC Series in Computing

Volume 77, 2021, Pages 1–12

Proceedings of ISCA 30th International Conference on Software Engineering and Data Engineering

Translating a Large Software from C# to C++: An Experience Report

Edgar Gabriel, Nicholas Roos Biddle, Shane Tran Whitmire, and Allen Casey

Parallel Software Technologies Laboratory,
Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
egabriel@uh.edu, nrbiddle@gmail.com, whitmireshane@gmail.com, allencasey337@gmail.com

Abstract

Converting source code from one programming language to another is a problem that occurs regularly in real life, but has attracted limited attention and has not been investigated systematically. This paper presents the challenges of translating a large source code from one high-level programming language to another, and the solutions developed to resolve these problems. Furthermore, the paper introduces a systematic classification of the occurring challenges during the translation process and elaborates on the ability to handle them without manual intervention by a programmer. A detailed performance comparison between the two language versions of the code is being performed.

1 Introduction

Proteomics refers to the study of proteins produced by an organism. Understanding protein structures and protein-protein interactions is essential for understanding their function within an organism. Mass Spectrometry (MS) is an analytical technique that provides information about protein structures. A Mass Spectrometer breaks a protein into smaller parts (so called ion fragments) and records the mass-to-charge ratio and relative intensity of these fragments. A software tool is subsequently used to identify the proteins present in the sample by comparing the measurements generated by the instrument to public databases of known proteins. The result of the protein identification step is typically a list of possible protein matches, ranked and sorted based on a given criteria (e.g. number of matching intensity values between the observation and a candidate protein). Since a single protein generates a large number of ion fragments during an experiment, identifying a protein can be compute and data intensive: a single sample can result in tens of Gigabytes of raw data produced by the MS. If there is no a priori knowledge about the organism that the sample was taken from (e.g. in forensic analysis), the databases used for the identification can be very large and contain millions of candidate proteins. Furthermore, hundreds or even thousands of samples have to be analyzed to reach statistically relevant conclusions in some clinical studies, introducing time constraints for the data processing. Thus, the required computational and memory capacity of standard desktop systems are often insufficient to meet the needs of the analysis.

In this project we aim to develop a parallel software package for protein identification suitable for utilization on high performance computing (HPC) clusters and/or cloud computing resources. Generally speaking, an application benefits from using a parallel compute resource by dividing a (large) problem into smaller parts/tasks and assigning different compute resources for each task. In our scenario this can be achieved by either splitting the input data (i.e. the data generated by the mass spectrometer) into multiple files and processing them separately, splitting a very large database file into multiple independent databases, or a combination of both approaches. However, some scenarios require further processing of the results of the independent tasks to determine the overall result. For example, if the parallelism was achieved by splitting the database file used for protein identification, establishing the final list of best-matching proteins will require an additional ranking/sorting across all partial results generated by the independent tasks. In fact, a number of algorithms used internally during protein identification are *not trivially parallel*. In other words, the result of the parallel execution would not create the correct result if one would simply deploy multiple instances of the sequential code simultaneously. Consequently, using HPC or cloud computing for protein identification requires the development of a parallel application that supports communication of partial results between tasks, e.g. using the Message Passing Interface (MPI) [10]. To the best of our knowledge and based on our literature review, there is as of today no fully parallel software for protein identification available that fulfills these requirements.

The starting point of our work is MetaMorpheus, an open source proteomics software package used by numerous scientists in the domain [8]. The code is written in C#, which is a powerful programming language for desktop systems, but is not a language well-suited for HPC environments. As of today, none of the popular parallel programming paradigms used in HPC (e.g. MPI [10], OpenMP [13], Spark [18]) support C#. Thus, the first step in this project consists of translating the C# source code to a programming language that is more widely used in HPC environments, such as C++.

Converting source code from one language to another is a problem that has not been investigated systematically, despite the fact that it is a regularly occurring issue. Numerous companies were looking into translating their proprietary code into a newer language in the late 1990's to cope with the Y2K problem [9]. This challenge arises today if there is a significant architectural change for an application, such as porting the application from desktop systems to mobile devices, or switching from local processing to a cloud-based back-end in the application. There are very few tools to assist in the translation process [2, 14, 15], and hence, most companies choose to reimplement an application instead of translating existing source code.

Compilers translate the source code of an application from a high-level programming language into the programming language understood by a processor, known as machine instructions. Source-to-source translation between two high-level programming languages is generally considered practical for very limited use-cases as of today, such as inserting compiler directives to support compute accelerators [7, 17]. More recently, IBM published as part of project CodeNet [5] a large dataset containing more than 500 million lines of code in multiple programming languages. The dataset has been used to train a machine learning model to perform source-to-source translation between two high-level languages with some success. The approach taken in our work includes using a tool that assists in the translation process as well as manual code adjustments and translation.

The contributions of this paper are threefold. First, the paper presents our approach to translate a large source code from C# to C++, and details the problems and challenges faced during this process. Second, we perform a detailed performance comparison between the C# and C++ version of the code. Third, we classify the challenges occurring when translating non-

trivial source code between two high-level languages, and elaborate on the ability to handle them without manual intervention by a programmer.

The remainder of the paper is organized as follows: Section 2 provides some necessary background information. Section 3 gives technical details about the conversion process, problems faced, and the solutions deployed during the conversion. Section 4 provides quantitative metrics comparing the original C# and the new C++ version of the code, as well as performance numbers for some use-case scenarios. We provide a systematic discussion and classification of the challenges in Section 5. Finally, Section 6 summarizes the paper and gives on outlook on the future work in this domain.

2 Background

C# Programming Language C# was built as an extension of the C language. The language is designed based on what is known as managed code, i.e. code that is *managed by a runtime* [1]. For C#, the managed code is in Common Intermediate Language (CIL), and the runtime is the .NET Common Language Runtime (CLR). The CLR interprets the managed code and converts it to machine code using just-in-time compilation. The runtime further provides support for features such as memory management, thread management, type safety, exception handling, and garbage collection. While these features offer many benefits, they introduce overhead that cause performance to suffer. Though C# was originally designed for use on Windows systems, C# code can also be executed on Linux systems using tools like the dotnet environment provided by Microsoft [12].

C++ Programming Language C++ was initially developed in the mid 1980's as an extension to the C language [16] incorporating object-oriented capability such as objects, inheritance, encapsulation, and polymorphism. Since 1998, it has been a standardized language maintained by the International Organization for Standardization. C++ is upwards compatible with C, and is a strongly-typed language. The language also allows for low-level memory manipulation, but mandates manual memory management by the application developer.

C++ is a compiled language. The build pipeline for C++ includes pre-processing, compiling, and linking. In general, C++ source files are compiled into object files. The object files are then linked together to produce an executable. The compiler generally checks for language rules at compile time, while the linker ensures that all names and functions used are correctly resolved. Few, if any, checks are performed at run-time, and if run-time checking is desired, tests must be written into the source code by the programmer. The compile-link-execute pipeline can lead to very fast performance if everything is optimized properly, making C++ one of the preferred languages for high performance computing.

3 Contributions

This section details the process of translating MetaMorpheus [8] from C# to C++, as well as some of the problems and challenges encountered during the process. The application has numerous dependencies to other libraries and software packages. One of them, MzLib, had to be translated using the same tool and approach as MetaMorpheus itself. For the sake of simplicity, we will not distinguish between MetaMorpheus and MzLib in the subsequent discussion, but treat them as a single, large software package. Other dependencies have been

resolved by identifying C++ libraries offering similar functionality, or by re-implementing the required functionality in C++ (this option was typically used for smaller routines).

The initial conversion step was done using the C# to C++ converter by Tangible Solutions [15]. This commercial tool is one of only a very small number of tools available to support end-users in such a project. The tool can convert individual files, directories, and full Visual Studio projects from one language to another. The software tool created a starting point that saved many Person Months of work by mapping C# classes and their elements to C++ classes, converting high level data structures used in C# (e.g. `Arrays, HashSet`) to their C++ counterparts (`std::vector, std::set`), and adjusting the syntax of many generic code sequences, e.g. loops. The tool allows the user to make choices about the code being generated, e.g. by selecting whether to use the regular `std::string` in the resulting C++ code vs. `std::wstring` wide strings, or various source code formatting options. Furthermore, the user can select the C++ standard that the generated code will use. We opted for the most recent official C++ version supported by the tool at the time of development, C++ 2017, since it allowed for easier mapping of some C# features used by the code to their C++ equivalents such as `nullable` C# variables to `std::optional` in C++17.

While the code generated by the conversion tool is a good starting point, very few of the automatically generated C++ source files could be compiled without additional manual work. Some functions had to be translated fully manually, since the tool did not support translating them directly e.g. local C# functions, or C# functions returning tuples of objects. The next step therefore involved creating the compilation-related infrastructure (Makefiles, configure script, etc.) and making the required changes to compile each file individually.

In the following, we will discuss the most relevant problems that we encountered during the translation process, and the associated solutions developed for this software package.

1. **Memory management**: C# utilizes garbage collection for memory management, while C++ requires the manual insertion of `delete` statements for freeing unused dynamic objects. The automatic conversion tool inserted `delete` statements when it was possible to deduce that the object was going out of scope, and otherwise left comments indicating that it could not free the object. Nevertheless, identifying where to free an object in a large software package in which objects are passed through multiple software layers is non-trivial, especially if one would like to avoid creating numerous copies of the same object.

   One option discussed early in the process was to utilize C++ smart pointers (e.g. `std::unique_pointer`). A smart pointer automatically releases the memory that the pointer is pointing to after the last reference to the object goes out of context. This solution was ultimately dismissed for multiple reasons. First, since the automatic conversion tool did not offer an option to use `std::unique_pointer` during the code translation, this would have required changing most of the source code retroactively. Second, smart pointers come with a processing overhead compared to regular pointers, which is counter productive if the stated goal of the work is to maximize performance (see also section 4 for some details). Finally, since our ultimate goal is to develop a parallel version of this code, none of the targeted parallel programming models (MPI, OpenMP, CUDA) support passing objects using the `std::unique_pointer` construct. As an example, it is unclear whether `std::unique_pointer` could handle asynchronous communication using MPI, since the buffer pointer is passed from the compiler's perspective only during the initiation of the communication operation. This would mean the object could not be released until the ongoing communication in the background completed.

2. **Initialization of static class members**: a number of classes in the software package have static member variables, such as a dictionary for storing the chemical elements of the periodic table. A problem arises from the fact that C++ does not allow the programmer to define an order in which static members are initialized across multiple classes.C# on the other hand has the notion of static constructors and establishes rules for the invocation of static constructors. It is beyond the scope of this paper to discuss pros and cons of both approaches (and there are heated debates in various forums about this topic). The purpose of this paper is to merely point out that this is a significant problem when translating a C# code to C++. The solution applied in our case was to create a generic initialization routine that has to be invoked very early (ideally the first function) in the code, which will ensure that the static class members are set up in a well-defined order in the C++ version.

3. **C# library functions**: C# supports numerous functions that are not available in C++. Consider for example the `ToString()` method which converts an object to its string representation in C#. This function does not represent a major challenge for most classes, since the C# code overrides the inherited `ToString()` base method. Thus, providing the C++ version of that method does not differ from translating any other code. However, there are instances where the solution is more complicated. For example, while C++ supports `enum` classes and hence provides a mechanism to translate the corresponding C# constructs, an enum class in itself cannot contain any further methods in C++. This means that supporting the `ToString()` method for an enum class requires either subclassing the enum class, defining static functions to convert an enum value to a string, or casting the enum value e.g. to an integer (which does not inherently indicate to an outsider the value of the enum object, defying the original purpose of the `ToString()` method). We decided ultimately to use static functions converting each value of an enum class to a string (and vice versa) and to implement these functions along with the enum classes in order to to maintain readability and maintainability of the source code.

   It should also be noted that the efforts required to provide the equivalent C++ code for some C# built-in functions can be quite substantial, e.g. developing a C++ counterpart to the C# `Enumerable.GroupBy()` method.

4. **Syntactical discrepancies between C# methods and their C++ counterparts**: in some instances mapping a method used in the C# code to a C++ function seems straight forward, only to realize later some subtle differences in the syntax. For example, the C# `List.BinarySearch()` method returns the index of a value if found in a sorted list. Similar functionality is provided by `std::binary_search` in C++. The technical difficulties stem however from the different behaviour of the C# and C++ functions in the case that the element is not found in the list. The C# version returns a negative number that is the bitwise complement of the index of the closest element in the list, while the C++ version uses a boolean value to indicate that no matching element was found. Since the original code made extensive use of the C# syntax, we ultimately ended up re-implementing the binary search function in C++ emulating the C# syntax, and not using the C++ `std::binary_search` method.

3.1 File I/O

An integral part of most scientific applications is managing input and output files. In the case of a proteomics application this includes, but is not limited to, configuration files, spectra (data)

files from the instrument, database files, and some generic information such as the periodic table, etc. For the sake of the following discussion we will distinguish between file formats that are based on text files, and file formats that use an xml format.

Text file formats: Text file formats are in most instances relatively straight forward to translate from C# to C++. In the following we discuss some of the file formats used by MetaMorpheus and the solutions applied for the C++ code.

- **Toml files**: toml (Tom's Obvious, Minimal Language) files are used as configuration files in MetaMorpheus to describe, for example, what task should be performed by MetaMorpheus (e.g. Search Task, Calibration Task) as well as parameters of the task to be performed. File sizes are typically small, and there are multiple C++ software solutions for reading and writing toml files. We chose for our code the tinyToml [6] header only library.
- **Mgf files**: The mascot generic format (mgf) is a text-based file format to store spectral data generated by a mass spectrometry instrument. Mgf files can be very large, reaching hundreds of Megabytes or even Gigabytes in size. Characteristics of each ion fragment are stored between `BEGIN` and `END` statements, making the file format easy to parse. Only minimal changes were required to the code generated by the Tangible Solution C# to C++ converter. However, there is room for performance improvements in the code.
- **fasta files**: this is a popular file format used to store protein databases. The main complexity when reading a fasta file stems from parsing the first line for each protein, which contains metadata about the protein. Parsing is typically done using regular expressions (regex). While there are some differences in the regex interfaces between the C# and C++ versions, the regex expression of the C# code – which is the most challenging aspect when dealing with regex functionality – could be reused in most instances in the C++ solution.

xml-based file formats Although xml-based file formats can also be considered text files, the structure of these files is typically significantly more complex than the formats discussed above, necessitating different solutions for reading and writing these files. As a representative example of xml file formats used by MetaMorpheus we will focus on the `mzML` file format. This file format is used to store mass spectrometry data, including the Spectra (i.e. the tuple of mass-to-charge ratios of the fragments observed and their intensity values), but also important metadata, such as the instrument used for data generation, date of data generation, etc. The Proteomics Standard Initiative [4] maintains the most up-to-date version of the xml schema definition (xsd) files for this file format. The xsd file itself is over $1,100$ lines, indicating that the mzML file format is not trivial. The C# version of MetaMorpheus used the Microsoft `xsd` tool to generate the code to read and write mzML files. Therefore, handling a file format such as mzML typically consists of two different types of code from the software perspective: code that populates the generated data structures of the Microsoft `xsd` tool, and the actual read and write routines automatically generated by a tool such as `xsd`.

One of the main challenges during the code conversion process was deciding how to approach the conversion of the C# xml read and write functionalities to C++. Three possibilities have been considered:

1. the Tangible Solution C# to C++ translation tool did also generate C++ versions of the Microsoft `xsd` C# read/write functionality;

2. using functions extracted from another C++ proteomics library to read and write mzML files, and write wrapper functions to map the data structures used from the third party library to our software;

3. using a tool to generate C++ code from the xml schema definition file.

Option 1 was dismissed based on the fact that the C# mzML read and write functions generated by the Microsoft `xsd` tool also used functionality in the `System.Runtime.Serialization` package that is not open source or publicly available. Option 2 was ultimately also dismissed due to the fact that the abstractions used by the third party library and MetaMorpheus were sufficiently different that a direct mapping of classes and objects was not easily implemented, and would have required significant efforts in understanding the third party library itself.

Ultimately, it was decided that using a code generator based on the available xml schema definition file would be the best approach for this problem. Among the publicly available tools to fulfill this task, the Code Synthesis `xsdcxx` tool and serialization library was chosen since it is available on most modern Linux distributions. The resulting source code generated by `xsdcxx` is very large, and not necessarily easy to read. Therefore, the main effort on our part was related to the functions in MetaMorpheus that invoke the `xsdcxx`-generated functions.

To highlight an additional challenge faced in this task, note that the mzML file specification supports binary data. Raw data typically consists of single- or double-precision floating point values, and has to be converted first to a base64 representation according to the mzML specification. In addition, binary data arrays can also be compressed using the zlib compression library. These steps required manual coding for the C++ version, since the code generated by the C# to C++ converter did not provide an adequate solution for this scenario.

3.2 Unit Tests

Translating unit tests requires additional time and resources, but the benefit of having a set of tests that allows the programmer to verify the correctness of the translated code is invaluable in the overall process. The biggest challenge for this part of our work stems from the fact that C# has support for unit tests as part of the `.NET Core` environment, while C++ does not provide anything even remotely related. The closest technical solution would be the Google Test software [3] which is based conceptually on the xUnit framework [11]. We opted ultimately for a different solution, which kept the grouping of unit tests similar to the C# organization on a per-file basis, but added an individual main function to each test file. Shell scripts are used to automate the execution of the unit tests.

4 Evaluations

4.1 Code Statistics

In the following, we present some source code statistics comparing the C# and the C++ version(s) of the code. The numbers shown in table 1 represent various sums of MetaMorpheus and MzLib combined, excluding the source code for the Graphical User Interface in the C# version (which was not converted for the C++ version) and the unit tests.

Data for the C# version are shown in Row 1. Column 1 contains the number of files that were written by the authors of the C# code, while Column 2 shows the total number of lines of that code. Columns 3 and 4 present the number of files and the number of lines generated by the Microsoft `xsd` tool to read and write xml-based file formats (e.g. mzML, pepXML).

Table 1: Code Statistics of the C# and C++ version

	No. of files	No. of lines	No. of generated files (xsd/xsdcxx)	No. of generated lines (xsd/xsdcxx)
C# original code	207	32,072	6	27,154
C++ from translation tool	418	51,397	682	42,064
C++ after manual fixes	464	72,883	16	174,558

Row 2 contains the same metrics based on the source code created by the C# to C++ automatic translation step, while Row 3 contains the statistics for the working C++ version of the code. The most interesting data in Row 2 are shown in Columns 3 and 4. The tool from Tangible Software Solutions converted the files generated by the Microsoft `xsd` as well, and it seems that an internal heuristic was triggered to create separate files for each class in the xsd specification, hence the large number of files (682 C++ files) generated out of just 6 C# files. However, since the code was not functional, these files were replaced in the final version with the solution described in Section 3.1 using code generated by the `xsdcxx` toolkit. This version reduced the number of files significantly (from 682 in the initial C++ version to 16) but increased the number of lines dramatically to over $170,000$. This code did not require any modifications and work on our side, however.

The key finding of this analysis is that the C++ solution has more than twice the number of files of the C# version, which can partially be explained by the fact that for each C# file, two C++ files are generated: one header file and one source code. There are still 50 additional C++ files that are the result of implementing C# built-in functionalities and for providing other dependencies that were not detailed in this paper for the sake of brevity. The current C++ solution contains a total of 480 files with more than $246,000$ lines of code.

4.2 File I/O Benchmarks

As discussed previously, file I/O is an integral part of the application overall. Therefore, we evaluate the performance of file I/O operations using a text-based file format (mgf files) and an xml based file format (mzML) for the C# and the C++ version. In the case of the mgf file, the application itself is only utilizing read operations for this file format, and hence only performance numbers for read operations are presented.

The file I/O benchmarks have been executed on the *salmon* server at the home institution of the authors. This server has a single Intel W3565 3.20GHz processor with 4 cores (8 threads) with 24GB of main memory. The system is running OpenSuse 15.2 as the operating system, g++ version 10.2 for the C++ version, and Microsoft dotnet 3.1.7 for the C# version. All tests have been executed 5 times, and the average of the five measurements is being presented. Since the system was used in a dedicated mode for these measurements, the variance between the measurements was negligible.

The left part of fig. 1 shows the performance numbers obtained for the mgf files. Two files were used for these measurements, 112MB and 197MB in size. Overall, there was minimal difference in the performance between the C# and the C++ version in these tests, with each language version being slightly faster in one of the two scenarios.

The results are more interesting for the mzML files. Recall that reading and writing mzML files utilizes manually written code as well as automatically generated code, using the `xsd` tool in the C# version and `xsdcxx` in the C++ version (see 3.1). Three different test files were

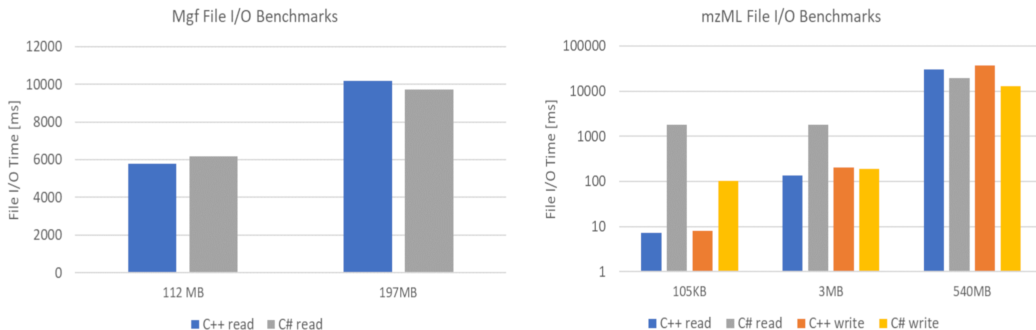

Figure 1: Performance results for reading mgf files(left) and mzML files (right) on the salmon server using the C++ and the C# version of the code.

used in the subsequent tests: a small file 105 KB in size, a medium file approximately 3MB size, and a large 540MB file. The benchmarks for this file format measured the read and write time for each file separately.

The results indicate a significant performance advantage for the C++ version for both read and write operations over the C# version when using the small file, since the costs of invoking the just-in-time compilation by the dotnet runtime environment are dominating the execution time for this test case. The performance advantage of the C++ version is negated, however, when the file size increases. In fact, the C# version outperforms the C++ version by approximately 30% for the read operations (19.4 seconds vs. 30 seconds), and by approximately 55% for write operations (12.9 seconds vs. 37.1 seconds) for the largest file size. Our investigations revealed that the vast majority of the time is lost in the code generated by the `xsdcxx` tool. This is due to the fact that the code generated by the `xsdcxx` is using C++ smart pointers (i.e. `std::unique_pointer`) for its data structure, which come with a significant processing overhead. To demonstrate the processing overhead of smart pointers, consider the left part of fig. 2 which shows the execution time of a vector add operation using standard (raw) pointers vs. a `unique_pointer`. The results indicate that using `unique_pointer` in a compute intensive operation imposes an 800% overhead on the *salmon* server when using this feature.

Furthermore, it was interesting to denote that the C# version was outperforming the C++ version only if the resulting output file did not already exist. The execution time of the C# version increases from 12.9 seconds to over 40 seconds for exactly the same file if the application is overwriting an existing file. In contrast to that, the C++ version does not show any performance degradation in this scenario.

4.3 Application Benchmarks

Next, we present performance results obtained by running a full application scenario. While MetaMorpheus can be used for a variety of analysis tasks, the focus in the following tests is on a protein identification for cross-linked mass spectrometry data, since this is the motivating use-case for this project.

Two data sets are used in the following tests. The first one, BSADSO200, is based on a 112 MB mgf Spectra file and a very small database file with data from a single protein. The second data set, referred to as `Ribosome` in the following discussion, uses a 197MB mgf Spectra file, and

a more realistic (though still small) database containing data of 66 proteins for identification.

In addition to the *salmon* server described above, tests were also executed on a compute node of the *sabine* cluster at our institution. A compute node of this cluster is representative of a high-end server, with two Intel E5-2680v4 processors with 256 GB memory, a high-speed Intel Omnipath network interconnect (100Gb line rate) and a parallel Lustre file system. The Intel C++ 2019 compiler was used for the C++ version on *sabine*.

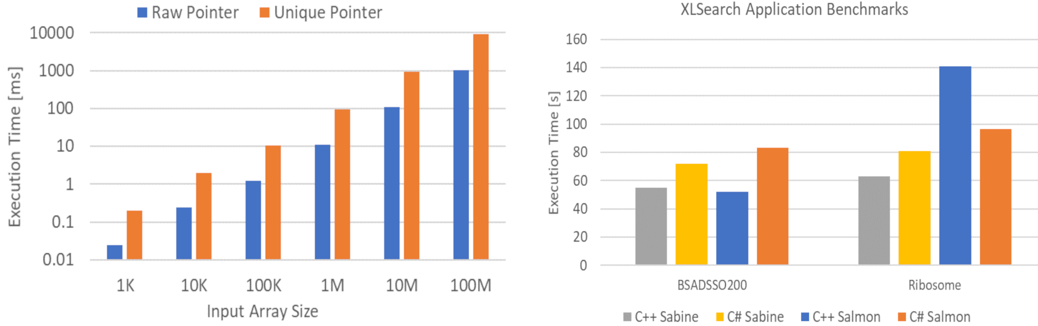

Figure 2: Results evaluating performance of smart pointers vs. regular pointers (left) and for a full application use-case (right).

The results of the application benchmarks are shown in the right part of fig. 2. It has been verified that the result of the C++ version and the C# version match for both data sets. On *sabine*, the C++ version outperforms the C# version in both application scenarios by approx. 22%. On the *salmon* server however, the C++ version outperforms the C# version for the BSADSO200 testcase, but not for the Ribosome data set. The performance penalty in the latter scenario stems from the larger memory footprint of the C++ version due the ongoing work related to the memory management (see Section 3). This ultimately results in the kernel having to swap memory to disk for this testcase on the *salmon* server. However, based on the data from *sabine*, we would expect the C++ version to be faster than the C# version on *salmon* in the final version.

For the sake of clarity, we skipped over numerous aspects of the code conversion in this paper. It should be noted that, in addition to the points discussed here, there were some performance optimizations necessary in the C++ code in order to obtain this level of performance.

5 Discussion

In this section we would like to provide a generic discussion and classification of the challenges posed by the various functionalities of a real-life application when translating it from one high-level language to another. Ideally, the goal would be to have an automated software tool performing the translation and only minimal intervention from programmers would be required.

1. *Capturing syntactical differences between languages*: this aspect includes translating the syntax of a language (e.g. variable and class declarations, functions, loops, etc). While this part can be automatized to a large extent, we provided two examples (memory management and static constructors) that seem very challenging to resolve without manual

intervention. Note that this might not be a symmetric problem, e.g. translating from a language using explicit memory management to a language using garbage collection is easier than vice versa.

2. *Translating functionality provided by standard libraries*: many/most languages provide libraries to support common data structures and algorithms. While these libraries are not necessarily part of the language specification itself, they are often managed and defined by the same governing body as the language standard to ensure cross-platform availability and portability. From the high level perspective, providing support for translating the functionality of a support library seems manageable, since the problem is limited to a potentially large but fixed and well-defined set of functions. However, we provided one example that highlights the challenges in this category. There are many applications for which mapping the C# `Enumarator.BinarySearch` to the C++ `std::binary_search()` functionality is valid, but this is not always the case. An automatic translation tool would have to be able to capture differences in the expected behavior of the functions and/or the application context in order to decide the correctness of mapping a function from one support library to another.

3. *Translating third-party libraries*: third-party libraries are being developed without control of the governing body managing a programming language. A reasonably complex real-life application will typically have dependencies to multiple third-party libraries. If the source code of a dependent library is available, it could be included and translated along with the target application. However, the source code of some libraries might not be publicly available and hence cannot be translated, in which case an alternative library providing the same or similar functionality in the target language has to be identified. The latter case will most likely require manual intervention to adjust the application to the new library.

4. *Tests*: it is invaluable to be able to translate (simple) unit tests and verify the correctness of small sections of the translated code before moving to larger and more complex scenarios. Challenges in managing unit tests often stem from output formatting, e.g. encoding of strings, floating point value representations, and the number of digits displayed. Based on our experience, deciding whether a unit test in the translated language is declared to have passed or failed will in most instances require human verification early on.

With sufficient resources, most challenges discussed above can probably be tackled by an automated tool for a fixed set of languages, but eliminating all interventions by programmers is most likely not possible within the near future.

6 Summary and Ongoing Work

This paper discussed challenges of translating a real-life application from one high level programming language (C#) to another language (C++), and presented the solutions developed during the process. The paper provided details about the manual code modifications that had to be made for generating working C++ code, code statistics comparing the C# and C++ version of the code, and an evaluation of the performance for some components and application use-cases for both versions.

Since the overall goal of this project is to develop a version of MetaMorpheus for HPC systems capable of processing larger data sets than those possible on desktop systems, currently

ongoing work focuses on the development of parallel versions of the C++ code using the Message Passing Interface (MPI). Multi-threaded parallelism using `OpenMP` directives has already been incorporated in some code sections. We omitted discussion on this topic in order to keep the paper focused on the software translation aspects of the work.

Acknowledgments. The authors acknowledge the use of the Sabine Cluster from the Research Computing Data Core at the University of Houston to carry parts of the research presented here.

References

[1] Ben Albahari. A comparative overview of C#. *Web link: http://genamics. com/developer/c-sharp_comparative. htm*, 1, 2000.

[2] Alex Albalá, Juan Lopez, and Gerard Solé. AlterNative: Human-Like translations from .NET assemblies to C++. https://alexalbala.github.io/Alter-Native/, 2021.

[3] Google. Google test: Google's c++ test framework! https://github.com/google/googletest, 2020.

[4] Hupo Proteomics Standards Initiative. mzml 1.1.0 specification. http://www.psidev.info/mzML.

[5] IBM. Kickstarting AI for Code: Introducing IBM's Project CodeNet. "https://research.ibm.com/blog/codenet-ai-for-code", 2021.

[6] Shinya Kawanaka. tinytoml: A header only C++11 library for parsing TOML. https://github.com/mayah/tinytoml.

[7] Dan Li, Haijun Cao, Xiaoshe Dong, and Bao Zhang. Gpu-s2s: a compiler for source-to-source translation on GPU. In *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, pages 144–148. IEEE, 2010.

[8] Lei Lu, Robert J Millikin, Stefan K Solntsev, Zach Rolfs, Mark Scalf, Michael R Shortreed, and Lloyd M Smith. Identification of MS-cleavable and noncleavable chemically cross-linked peptides with MetaMorpheus. *Journal of proteome research*, 17(7):2370–2376, 2018.

[9] Mark Manion and William M Evan. The y2k problem and professional responsibility: a retrospective analysis. *Technology in Society*, 22(3):361–387, 2000.

[10] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard Version 3.1*, June 2015. http://www.mpi-forum.org.

[11] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[12] Microsoft. Microsoft .NET. https://dotnet.microsoft.com/download.

[13] OpenMP Application Review Board. *OpenMP Application Program Interface, Draft 3.0*, October 2007.

[14] Usman Sarfraz. Auto Port C# Codebase to C++ – csPorter Newsletter February 2019. https://blog.codeporting.com/2019/02/01/auto-port-c-codebase-to-c-csporter-newsletter-february-2019/, 2019.

[15] Tangible Software Solutions. C# to C++ converter. https://www.tangiblesoftwaresolutions.com/product_details/csharp_to_cplusplus_converter_details.html, 2019.

[16] Bjarne Stroustrup. An overview of C++. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming*, pages 7–18, 1986.

[17] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. A source-to-source OpenACC compiler for CUDA. In *European Conference on Parallel Processing*, pages 178–187. Springer, 2013.

[18] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.