Porting LHC software to 64-bits

Antonio Vilela Pereira
Supervisor: Rosa Maria Garcia Rioja

August 2005

Abstract

It's given a summary of the activities performed by the author as a summer student for the CERN OPENLAB summer student programme, consisting of porting the SEAL and POOL packages, from the LHC software stack, to 64-bit architecture machines.

Email:
Antonio.Vilela.Pereira@cern.ch, vilela@uerj.br

1 Introduction

As 64-bit architectures emerge as performance gaining possibilities (most obvious, increased virtual memory addressing space), one has to consider how to adapt to this major change of paragigm and, of course, all other specific properties. But what happens if all that is needed is to run an existing application working in a regular 32-bit architecture, is that necessary to rewrite the entire code? The answer is yes and no. If you really want to adress these specific new properties and options that may not be standardized, unless you are a kind of prophet, there is not much to do, however, it would be very nice, not to say crutial to be able to recompile existing written code in these machines without any additional modifications. The good news is that it is just like that, as long as your code is *clean*. That means that you basically have to get rid of direct assumptions on your 32-bit architecture and correct the same errors that you would normally do, but that just don't make a difference in 32-bits.

So, as everything has two faces, the bad news is that it is possible and all you have to do is get started and try to get used to some recurring types of *mistakes*. This report is about the experience trying to do this *porting* of two of the base packages, SEAL and POOL, in what one may call the LHC software stack, to a 64-bit architecture, specifically AMD64, not that it should make a difference if one would compile it, for example, in IA64.

After this short introduction, I'll cover some general porting issues and procedures, then discuss the SEAL and POOL packages, not in much detail, just to contextualize things a bit. Then I'll talk about this management and building tool, or framework, SCRAM, used widespread in LHC software development and implementation, followed by a more detailed description of the activities and problems encountered by the author in this period.

2 Porting Considerations

Disregarding any assembly code you might wanna port to 64-bit architectures and if you're using some standardized high level language, then the task of porting generally simplifies to some common issues in your code.

The first thing you should be aware of is what data model is used in this kind of architectures, which are the data sizes. While **int** remains 32-bits, **long** (in Linux) is now 64-bits, the same thing with the pointer size and also with types like **size_t**, **time_t**, etc... Table 1 shows the basic data types , with their sizes in bytes and alignment.

You can probably see that even if you successfully mixed and casted among data types in 32-bits, this might not be the case in 64-bits. This is the simplest and maybe because of that, the most common source of porting issues. Hence, a few words about casting.

- Integer Widening

  If you are casting an integer to a larger size, the compiler will preserve the sign by filling the the new leftmost bits with ones if the value is negative and with zeros if the value is positive. If it converts to an unsigned integer, the new bits are filled with zeros always.

- Casting to Smaller Type

Type	Size	Alignment
char	1	1
short	2	2
int	4	4
long	8	8
long long	8	8
float	4	4
double	8	8
long double	10(12)	16
void *	8	8

Table 1: Data model for 64-bit architectures

If you cast to a smaller type the bits in excess will be discarded, a **long** to an **int** in 64-bits, for example. The same happens when casting pointers to integers. If you cast to a signed type and after the bits are discarded the leftmost bit is 1 then the number is negative, even if your previous one wasn't. If you cast to a unsigned type then this leftmost bit is just a part of the number.

- Signed to Unsigned

  If you are switching from signed to unsigned or vice versa there is just a change of interpretation from the compiler regarding the leftmost bit. If you are also changing the data size, then you can imagine it as two conversions in succession, first the widening or truncation and then the leftmost bit interpretation.

One might also be concerned on how big his structures will become and basically how they will lay out in memory, so issues like packing, padding and alignment become important. One might then want to rearrenge the data types in a structure (bigger to smaller data types is a general rule), or if you optimized your data structures to fit in a processor's cache for example, you would get some performance differences; again it's just an assumption (a more than valid one) on the specific architecture and you shouldn't expect this to be automatically transfered when passing to a new one. The mistake is not make the assumptions, just not knowing that you did them and that they might bring some issues. Of course this kind of modifications are only possible when you have enough freedom and flexibility to change the code, which is not always the case.

Generally if you follow some coding guidelines, you should have a smooth porting to 64-bits:

1. Don't mix types, specially if they have different sizes or one is signed and the other unsigned.
2. Don't cast pointers to **int** or even **long**, if you can avoid. There are secific types to use in these cases, like **uint_ptr_t**.
3. Use specific functions to access data sizes and systems parameters, like **sizeof()** or **offsetof()**.

4. Use preprocessor flags, like **__LP64__** (long and pointers are 64-bits) or **__WORDSIZE** (either 64 or 32).

5. Isolate your code from assembly implementations.

6. Don't use constants with #define, they might not have the same interpretation in both architectures.

7. Use specific compiler warnings for 64-bit issues, when available.

3 Packages - SEAL and POOL

- SEAL

  As a SEAL spokesperson might say, SEAL "aims to provide the software infrastructure, basic framework, libraries and tools that are common among the LHC experiments...select, integrate, develop and support foundation and utility class libraries...develop a coherent set of basic framework services to facilitate the integration of LCG and non-LCG software".

  In Table 2 you can find the SEAL packages, with a short description.

Foundation	Foundation and Utility Libraries and Plug-in Manager
MathLibs	Math Libraries Support and Coordination
Dictionary	LCG Object Dictionary
Framework	Component Model and Basic Framework services
Scripting	Scripting Services
Grid	Grid Services
Documentation	Education and Documentation

Table 2: SEAL package description

- POOL

  POOL is a common persistency framework for the LCG, with the capability of storing multi-Petabyte data and metadata in a distributed way. The idea is to deploy a multi-backend structure, with focus in maintainability and much more transparent data handling.

  POOL follows a kind of storage hierarchy, consisting of several layers of increasing granulanity. The first one is the **POOL Context**, which holds all objects a particular user can see. Now each context may reference objects only through a given **File Catalog**, which contains metadata to the physical files in a particular **Database**, which must have a general technology specification, such as RDBMS. The databases themselves are structured into **Containers** that group persistent **Objects** inside the database. Each container may differ in the *minor* technology type as long as it still holds to the *major* Database technology type.

POOL Context	Technology independent
File Catalog	Technology independent
Database	Major technology dependent
Container	Minor technology dependent
Object	Persistent Object

Table 3: POOL storage hierarchy

The package is structured in a component based architecture, with additional isolation provided by several services APIs. The three main component paradigms are the **File Catalog**, that cointains information about all the databases and makes the bridge between physical and logical filenames, the **Storage Service**, that can use the technology information in the catalog to dispacth operations to a specific storage manager that provides the translation capabilities necessary to make a persistent storage representation of a transient user object and the **Collection**, that performs the maintaining and management of the objects.

4 Building - SCRAM

SCRAM is a framework for distributing, configuring, managing, developping and building software for the LHC. The idea is to provide a transparent way for working with different environments, architectures and large number of external libraries and tools.

The first thing one may want to do with SCRAM is to create a project area. If you already have a particular release instaled on you system you may want just to install a small part of it for developping/using, so that SCRAM will look up for all the missing parts (libraries, binaries, etc...) in the release path and look for changes in your own project area. However, if it's not possible to use a pre-existing release, either because it's not avaiable in your system or, for example, it's not compiled for the desired architecture, you may just want to get (and compile) the whole code.

There are three main XML-type configuration files, that should be in a well known configuration directory, that define the project. The first file is the **BootStrapFile**, that defines how a project should be configured, with information like the project name and version, the name of the configuration directory in the tree, download locations and specifications, the source code directory and the **RequirementsDoc** file. This last one is the file that keeps track of all the external products, providing information for SCRAM to use when downloading the tools description; in this file you can have requirements for particular tools for all or specific architectures, defining the tools' name, version and configuration url. The third important project file is the **BuildFile**; while the first two are all about bootstraping and configuring the project, this one deals with building it. This is the first file parsed by SCRAM when performing a building operation, as it contains a sort of mapping between the directory tree location and build actions, described by defined makefiles. It should also contain general build rules for all or specific architectures, like which compiler to use, as well as includes to other specific files. You can also set specific rules to each subsystem or package by defining an apropriate BuildFile in each subdirectory in the project tree, which will be parsed successively by SCRAM.

To bootstrap a project you must first have some basic project configuration files (most important the BootStrapFile), which can be downloaded using CVS, for example. To create a project area one can use

```
scram project file:BootStrapFile
```

SCRAM will then set up the tools automatically and download whatever it was set to download; it will create a **config** directory, with all files used for bootstraping, as well as the build system files, downloaded from the CVS repository. The project information itself will be stored in the **.SCRAM** directory, and that's where it's going to look for project configuration. The actual tools' information, like versions and locations are stored in files, one for each tool, inside a subdirectory with the name of the architecture. Of course there are specific commands for installing, removing and setting the tools, which will basically modify these files just mentioned.

With the project created, you might want to build it and/or just use a particular binary or library you built or from your originating release. For this you have to set up the environment, like SCRAM definitions and general tools' locations. This is done with the command, for example if you're using **csh**

```
eval `scram runtime -csh`
```

The actual command is `scram runtime -csh`, wich gives a series of environment definitions to be evaluated with `eval`.

If you are building your project, what SCRAM does is the following. First it checks if the current directory is a project area, if not it returns an error. It then knows where to find the configuration files, which are read to create a compilation time makefile containing all the relevant paths in a **tmp** directory. It then parses the project **BuildFile**, as discussed before, creating also a correspondent makefile. Another makefile is created for the current directory and subsequently merged with the previous ones and also with any others defined for this particular location. It's worth to say that the makefiles and build products are created in the **tmp** directory following the same directory structure. The build then finally runs from the current directory down and the build products are moved to storage areas under the name of the architecture. The SCRAM command used to build is (The SCRAM commands can be abbreviated, like b stands for build, runt for runtime and so on)

```
scram b
```

There might be some make targets that can be added to this command and that should be found in a high level makefile in the **config** directory.

5 Dependencies

One of the tricky things about porting is that it's not only the piece of software you are working on that needs to be ported, but all its dependencies too, and there can be a large number of these. When they are well maintained, general purpose software, that can be easily done by recompiling it in whatever architecture you're working on. However, when these are parallel developping software and you're also talking about a software *stack*, fully porting your package means fully porting a lot others.

POOL is directly dependent on SEAL, as any other LHC package, directly or indirectly, but SEAL, as POOL, are the subject of the next section. In this section the idea is to enumerate the external software that were successfully compiled in AMD64 and also the ones who weren't.

The following external packages, either had already been compiled in AMD64 or had to be recompiled:

- boost (boost_1.32.0)
- cppunit (cppunit_1.8.0)
- cgsigsoap (cgsigsoap_2.6-1.1.9)
- gccxml (gccxml_0.60_patch2)
- mysql (mysql_4.0.24)
- mysql++ (mysql++_1.7.31_mysql.4.0.24)
- oval (oval_3_5_0)
- pcre (pcre_4.4)
- python (python_2.3.4)
- qmtest (qmtest_2.2.1)
- root (root_4.04.02(c)(f))
- sqlite (sqlite_3.2.1)
- uuid (uuid_1.32)
- valgrind (valgrind_2.4.0)
- xerces-c (xerces-c_2.3.0)
- unixodbc (unixodbc_2.2.6)
- myodbc (myodbc_3.51.06_mysql.4.0.24)

Most of the packages compile in a straightforward way, with the common sequence **configure**, **make**, **make install**. Some of the additional options can be checked in [1] for most of the packages. Just remember of course of the order in which certain packages have to be compiled, like mysql before mysql++ or myodbc and so on.

There are however some packages that couldn't be found or recompiled in 64-bits; this is the case of:

- oracle
- glite
- vdt
- lfc (could compile but with some issues)

Since this only has consequences on a single or few specific subsystems, those were left aside.

6 Compiling and Testing SEAL and POOL

SEAL had already been target of some considerable work, with some remaining issues. A CVS tag has been prepared called **SEAL_1_7_0-port64** for these porting efforts. Later on, the **SEAL_1_7_1** release, which should be somewhat the same thing, has also been patched following the above mentioned tag and used, this time focusing on compiling it with **icc**, while for the first one **gcc** was used as well as for the most part of the project.

When trying to install the project area as explained in a previous section, SCRAM will look for some tools descriptions that might not exist for your architecture. In this case it will just leave the tool description file empty, so what you have to do is install these tools yourself, or just edit these configuration files. You can check if SCRAM is *seeing* your tool description by typing

```
scram tool info TOOLNAME
```

Then to build the package, one should first set the environment

```
eval `scram runtime -csh`
```

After that you can just go the apropriate tree location (generally the **src** directory) and use the SCRAM build command

```
scram b release-build
```

where **release-build** is a make target (it can be found in the Project_makefile.mk) that does some additional steps. Before compiling again you might want to clean everything by using the target **release-reset-arch**, that will remove the logs, the tmp and product directory for your architecture.

The testing can be done using the **qmtest** tool. Just go into the qmtest directory (in my case, src/config/qmtest) and type qmtest gui

That will open a very friendly and self-explaining interface for performing the tests. With your environment set you can also just type the test application name anywhere and it will run. You can also use the OVAL tool, which will basically run the test and compare some chosen output to a reference one.

Table 4 shows the tests that don't succeed for **SEAL_1_7_0-port64** using **gcc** in AMD64, according to qmtest, since that's how one would like to compare to it.

It's just the qmtest output translated into a table. It's worth mention here that some of these unexpected standard errors and so on don't really seem to be failing, but it's good to keep it as a reference anyway and adress it after the more serious problem have been solved. For **SEAL_1_7_1**, the results relate, except for some, when compiling with **icc**, that don't compile, but the whole icc effort was intended just to check out some additional 64-bit related warnings, so this wasn't really addressed.

The POOL versions used were first the **POOL_2_1_1** and than the HEAD version checked out from CVS corresponding to **POOL_2_2_0**. As said by the POOL team, this last version is the one that is really going to be used in 64-bits and contained several modifications first pointed out by a previous summer student and that can be seen in [1]. It also implements a major change by using SEAL's subsystem Reflex instead of ReflectionBuilder for Dictionary operations.

To bootstrap the project, it's the same thing. Download the config files, use the bootstrap command from SCRAM, change the tools' configuration. For building it, those useful make targets don't exit *a priori*, so that you may just use the simple build SCRAM command and delete the files yourself or create yourself the targets, that doesn't really make a difference. For **POOL_2_1_1** the SEAL version used was the **SEAL_1_7_0-port64** while the **SEAL_1_7_1** version was used with **POOL_2_2_0**.

For testing POOL, I ended up using this OVAL tool, briefly explained above. For doing so just enter the **Tests** directory or a specific test and run

oval prod

This will compile, run, and compare to a specific reference output. You can see the latest run results by using

oval log

Anyway, it's a well documented tool, and one can find other options as the need comes.

The biggest problems up to the **POOL_2_1_1** release of POOL concernig 64-bits were in the most inner subsystems, **POOLCore**, **StorageSvc** and **RootStorageSvc** and this had already been dealt as explained in [1]. Basically, some constants badly defined mixed with 64-bit widened longs. The basic strategy that had already been performed and that I just followed was to change a lot of longs to ints and fix those constants. Of course a whole others "cast from pointer to integer of different size" and similar things had to be corrected.

In Table 5, I give a summary of the tests failing in **POOL_2_1_1** using **gcc**.

7 Conclusion

Porting to 64-bits is a task that could in principle be performed just by remembering some common issues concerning assumptions on your specific architectures. However, as bigger and inter-related your code is, and the less knowledge and flexibility you have to change it, more elaborated the problem becomes. Porting the LHC stack to 64-bits is a challenge that didn't start here and certainly didn't end either and needs full commitment, as time permits, also from the corresponding development teams to achieve and maintain a fully ported code.

References

[1] Michal Kapalka, *Porting and Optimizing LHC Software on Itanium 2 Platform*, 2004

[2] Dirk Duellmann, *The LCG POOL Project - General Overview and Project Structure*, Computing in High Energy and Nuclear Physics 2003, San Diego, 2003

cintex_tests.cintex	Unexpected exit_code, standard error
cintex_tests.persistency	Unexpected exit_code, standard error
cmsexamples_tests.dummy	No target matching linux
cmsexamples_tests.simhit	No target matching linux
dictionarygenerator_tests.complex	Unexpected standard error
dictionarygenerator_tests.simple	Program terminated by signal
fml_tests.fit_lifetime	Program terminated by signal
fml_tests.pyfml	Unexpected exit_code, standard error
minuit_tests.quad4fcppunit	Unexpected standard error
pylcgdict2_tests.pylcgdict	Unexpected exit_code, standard error
pyreflex_tests.basics	Program terminated by signal
reflectionbuilder_tests.allfunctionality	Unexpected standard error
reflectionbuilder_tests.dynamic_cast	Unexpected standard error
reflectionbuilder_tests.inheritance	Unexpected standard error
reflectionbuilder_tests.templates	Unexpected standard error
reflex_tests.reflex_unit	Unexpected standard error
reflex_tests.reflexbuilder_unit	Unexpected standard error
reflex_tests.simple1	Unexpected exit_code, standard error
reflex_tests.simple2	Unexpected standard error
sealbase_tests.debug01	Unexpected standard error
sealbase_tests.file06	Unexpected standard error
sealbase_tests.localserversocket01	Unexpected standard error
sealbase_tests.timeinfo01	Unexpected standard error
sealkernel_tests.exception	Unexpected standard error
sealkernel_tests.messagestream	Unexpected standard error
sealkernel_tests.properties	Unexpected standard error
sealroot_tests.sealrootdict	Unexpected exit_code, standard error
sealservices_tests.configurationservic	Unexpected standard error
sealservices_tests.dictionaryservice	Unexpected standard error
sealutil_tests.timers	Unexpected exit_code, standard error

Table 4: QMtest output for SEAL_1_7_0-port64 (gcc)

Collection_BackNavigate	Diffs., runs with some seg. faults
Collection_ExplicitReadPerformance	Diffs., runs with different values
Collection_ExplicitWritePerformance	Diffs., runs with different values
Collection_FileInfoRetrieve	Diffs., runs with different OVAL outputs
Collection_MultiFileUpdate	Diffs., runs with different OVAL outputs
Collection_Read	Diffs., runs with different OVAL outputs
Collection_Update	Diffs., runs with different OVAL outputs
DataRegression_DataSvc_CMSCollDataModel	Failed Run, cannot create catalog
DataRegression_DataSvc_SimpleEmbeddedRefs	Failed Run, cannot create catalog
DataRegression_RootCollection_BasicTest	Diffs., runs with different values
DataSvc_SimpleMultiBackends	Diffs., SEAL exception, different types
DataSvc_TypeChecking	Diffs., runs with different values
ExplicitCollection_Functionality	Diffs., runs with different results
MultiCollection_BasicFunctionality	Failed run, seg. fault

Table 5: OVAL tests summary for **POOL_2_1_1**