Porting and Optimizing LHC Software on Itanium 2 Platform

Michał Kapałka
(michal.kapalka@cern.ch, kapalka@plusnet.pl)

openlab internal report
Status: PUBLIC

Supervisor: Sverre Jarp

CERN
openlab for DataGrid applications
Developing Solutions for the Data-Intensive Science of the Large Hadron Collider

CERN, openlab, July–September 2004

Table of Contents

Part I

Optimizing the ROOT Framework for Itanium Architectures

1 Introduction

ROOT is an object-oriented data analysis framework. It is well described on its home page[1], so I will mention here only a few things about this framework, which are important from the point of view of my work. Firstly, ROOT is going to be used in the LHC experiment for data analysis. Therefore, its efficiency is essential, as the experiment is going to produce an enormous amount of data that has to be processed. Secondly, ROOT contains the Cint – a C++ interpreter, which speeds up the application development process. However, all the ROOT programs can be compiled to executable code, which obviously can make them run much faster. Of course, all the production applications for LHC will be in their compiled form, as their efficiency is really a crucial issue. Thirdly, ROOT has already been well ported to IA64 (Itanium) platform, so my task was not to make it run on these processors, but to make it run as fast as possible and to identify all the current bottlenecks and possibilities of future improvements.

Compilation of the ROOT framework is pretty straightforward. One just has to run the standard `configure` script with appropriate parameters and next – invoke the `make` program. So the steps would be, for ROOT v4.00.08a, the following:

```
tar -xzf root_v4.00.08a.source.tar.gz
cd root-v4-00-08a
./configure linuxia64gcc    # for GNU gcc compiler
or
./configure linuxia64ecc    # for Intel icc compiler
make
make install                # optional
```

Next, all the test programs can be built. They are in the `test` directory. Before compiling them, one has to set up the ROOT environment, e.g. using the `rootenv` shell script:

```
cd root-v4-00-08a
source rootenv
cd test
make name_of_the_test_program
```

The following four benchmarks, shipped with ROOT, have been used for the tests: `stress`, `bench`, `stressgeom` and `stressLinear`. They are all compiled executables, so their code is not interpreted by the Cint at runtime. To compile and

1 http://root.cern.ch

invoke them one has to type:

```
cd ROOT directory
source rootenv
cd test
make stress bench stressgeom stressLinear
./stress -b -q
./bench -b -q
./stressgeom
./stressLinear
```

All the benchmarks print their execution speed in ROOT Marks (RM). This unit is a relative speed as compared to the reference computer (Pentium IV 2.4 Ghz, 512 MB RAM) for which the number is equal to 600 RM.

The problem with the procedure described above is that if one wants to change compilation options, one has to edit the appropriate configuration files manually. These would be:

- `config/Makefile.linuxia64gcc` (for GNU compilers),
- `config/Makefile.linuxia64ecc` (for Intel compilers) and
- `test/Makefile.arch.`

The other thing is that with manual compilation doing a profile-guided optimization (see the next sections) requires much more effort. People's laziness make them create tools. I prepared a script which does a full single- or double-pass (profile-guided) compilation, runs the benchmarks and saves all the output in appropriate files. This makes it much easier to prepare a few different versions of ROOT and not to get lost in it. The syntax is the following:

```
rcomp gcc|icc|icpc s|pgo [compiler options]
where:
  gcc – use GNU C/C++ compiler (gcc)
  icc – use Intel compiler (icc)
  icpc – as above, but link with icpc, not with ld
  s – perform a single-pass (standard) compilation
  pgo – perform a two-pass (profile-guided) compilation
  compiler options – all the compilation options
```

By default, the script tries to compile ROOT version 4.00.08a. If you want to change it, set the `ROOT_VERSION` environmental variable before running it. The appropriate ROOT tar file – `root_v${ROOT_VERSION}.source.tar.gz` – has to be present in the directory from which the script is invoked. The ROOT build will be placed in the directory named after the ROOT version, compiler, compiler version and options, for example: `root-v4.00.08a-icc-8.0-PGO-O2-ftz-ipo-ansi_alias` (the "PGO" means "profile-guided optimization", so two-pass compilation). Please refer to Section 1 in Part IV to see what the script really does.

The tests were run on 1.5 GHz and 1.6 GHz Itanium 2 dual-cpu machines. To ensure that only single processor is used, what makes the results more objective, all the benchmarks were run by the `taskset` program.

The following sections will describe briefly the possibilities of code optimization of two modern Itanium compilers: GNU gcc and Intel icc.

2 Possible Optimizations

2.1 GNU Compiler (gcc)

GNU gcc is a very popular compiler, as it's free, mature, stable and well-maintained. It can optimize programs on many different levels – starting from machine-independent tree of code blocks, ending with platform-specific RTL instructions. However, although there are so many possibilities of tuning code generated by gcc, most people use only one of the two options: "`-O2`" or "`-O3`", sometimes having little idea about the differences between them. Fortunately, almost every kind of optimization used by gcc is turned on by either of them. "O2" optimizations are usually sufficient for most purposes. Most of them doesn't make resulting executables bigger, so it's always a safe bet. "O3" adds a few things to "O2", like function inlining. This may make the generated code bigger and much more difficult to debug. However, in cases when the speed really does matter, it may be worth a try, although it is not guaranteed that in all circumstances "O3" will produce faster code than "O2" does. For programs making complicated mathematical calculations, a few other optimization flags can be taken into account:

– `-ffast-math` – this should generate code for faster mathematical operations, but in some circumstances may be dangerous (i.e. when a compiled program "depends on an exact implementation of IEEE or ISO rules/specifications for math functions" (from gcc manual)),
– `-minline-int/float-divide-min-latency` – generates inlined code for integer or floating point divisions, using a latency-optimized algorithm (IA64-specific),
– `-minline-sqrt-min-throughput` – generates inlined code for square root operations, using a throughput-optimized algorithm (IA-64 and gcc 3.5 specific). Unfortunately, a corresponding latency-optimized version hasn't been implemented yet.

The other way of optimizing the code, which can be very fruitful on Itanium, is profile-guided optimization (PGO). It is a bit more complicated than a usual compilation, as it requires three steps:

– Firstly, a program has to be compiled with a special option (for gcc it is "`-fprofile-generate`"), so that its code is instrumented. This will allow for tracing the flow of execution of the program and save the information to local files.
– Secondly, the program has to be run, possibly doing all its usual tasks. As many

use cases as it's possible should be satisfied in the process, and thus it's usually desirable to make this procedure fully automated (e.g. by preparing appropriate benchmarks). The program will be running slower than usually, as it will write a lot of information about its execution flow.

– Last but not least, the program has to be recompiled with another option (for gcc it is "`-fprofile-use`") so that the compiler can use all the information generated in the second step and thus make its predictions more accurate.

Obviously, profile-guided optimization requires a significant amount of work. However, when the speed really does matter, this might be worth the effort.

2.2 Intel Compiler (icc)

The Intel icc compiler is a commercial product aimed towards Intel processors (IA32 and IA64). It is less frequently used than gcc, and there are at least two reasons for that: it costs some money (although it's free for non-commercial purposes) and still there are a lot of programs, especially for Linux, which use some gcc-specific constructs and just don't won't to compile with icc (see, for example, the discussions on Gentoo forums[2]). The Intel compiler is more and more compatible with gcc, but there are still programs which won't be able to compile with it. Fortunately, ROOT can use icc without any problems.

The Intel compiler supports all the major optimizations that are used by gcc. It also has the two basic compilation options: "`-O2`" and "`-O3`", which are well-suitable for most purposes. The former is usually a safe bet. The latter does things like loop unrolling and memory prefetching, which not only can result in larger executables, but also can be sometimes not desirable. The icc compiler supports also profile-guided optimization (PGO, see the previous subsection).

What is really unique about icc and what, at the same time, seems to be a very powerful technique, is multi-file inter-procedural optimization (IPO). Briefly speaking, this allows for inlining of code at binary (object) level. The idea is the following: to make a function or method inlined one usually has to put it in the same compilation unit as all references to it (i.e. in the same source file or in an included header file). When large programs, consisting of a number of shared libraries (e.g. ROOT), are considered, this is usually not achievable in a simple way. The IPO mechanisms create extended object files that contain some additional information and also extended shared libraries. When a program is to be linked, all its constituent object files and used shared libraries are examined and when the compiler finds out that some functions will be probably called very frequently and they are sufficiently small, it can make them inlined in the resulting executable. Obviously, this can result in larger files, but can pay off in many circumstances. It is also clear that IPO makes much more sense when combined with PGO that can provide better information on the usage of functions.

2 See, for example: http://forums.gentoo.org/viewtopic.php?t=113784

What is also worth noting is that Intel compilers make better use of the extra features of Itanium processors than gcc. They do much more advanced optimization, what can be easily seen in the generated intermediate assembly files. If it really makes difference will be shown in the next sections.

To sum it up, I would like to list the icc optimization options which I used for compiling the benchmarks:

– `-O2`/`-O3` – these are the standard optimizations described above,
– `-ftz` – flushes denormal results to zero; it makes mathematical operations run faster and usually should not make any harm,
– `-ipo` – turns on multi-file inter-procedural optimization (IPO),
– `-prof_gen`/`-prof_use`, `-prof_dir` – profile-guided optimization switches,
– `-mP2OPT_hlo_prefetch=false` – not documented flag which turns off memory prefetching (makes sense only with `-O3`),
– `-mP2OPT_hlo_loop_unroll=false` – not documented flag which turns off loop unrolling (makes sense only with `-O3`),
– `-ansi_alias` – informs the compiler that a compiled program is ANSI compliant (see icc documentation for details), what allows for performing more aggressive optimizations; this, however, implies a few assumptions and if either of them is not met by the compiled code, the generated executable may be invalid – so use it with care (you have been warned).

3 Experimental Results

3.1 Testbed

At the time of writing the document the newest release of ROOT was 4.00.08a. All the results are for this particular version, if not explicitly stated otherwise. The following compilers have been used:

– gcc 3.4.1, patched against the bug #16490 (PGO-related issue),
– gcc 3.5 (experimental), snapshot 2004-07-04, also patched against the bug,
– icc 8.0 (8.0.066_pl068.1).

Three dual-cpu Itanium 2 machines were used for the tests:

– oplapro35 – CPU 1.6 GHz, 2 GB RAM, Linux kernel 2.4.21-9.0.1.EL,
– oplapro50 – CPU 1.5 GHz, 2 GB RAM, Linux kernel 2.4.21-9.0.1.EL.cern,
– oplapro51 – CPU 1.5 GHz, 2 GB RAM, Linux kernel 2.6.7.

The first two nodes were used for benchmarking, the third one – for generating profiles of test programs (see the following section).

3.2 Compilation Problems

Itanium processors are very unique amongst others and they offer many new performance-improving possibilities. However, in this architecture most of the responsibility of code optimization has been put on compilers and so compiler developers have now really hard job. Itanium processors have to be explicitly informed about which instructions can be executed in parallel, what kind of speculation they can use and so on. Fortunately, their architecture gives quite a lot of possibilities for resolving uncertain situations and so the compilers task may seem easier. However, to use all these mechanisms new optimization algorithms have to be developed and obviously it can take some time until they have fully stable implementations. Therefore, although current GNU and Intel compilers for IA64 are already very mature products, forcing them to use very advanced optimization techniques may lead to compilation- or run-time problems.

The good thing about GNU compilers was that if they didn't fail at compilation time, they were producing correct executables. The Intel compiler, on the other hand, could make some of the benchmarks fail when very strong optimization was used. This doesn't necessarily mean that gcc is much more stable – it just indicates that the advanced optimization algorithms which are icc-specific should be used with care as they are laying somewhere on the Itanium compilers cutting edge, which has not been reached yet by gcc.

1) GNU gcc

As far as I remember there have always been problems with profile-guided optimization in gcc. This powerful technique, especially fruitful on Itanium, seems to be still not very well implemented in this compiler. It works well for simple examples, but fails on complicated packages, like ROOT. At the beginning of my work this year the problem looked like the following: the first compilation pass was ok, all the benchmarks were generating profiling information, but the second compilation pass failed because of the compiler crashing with the well-known "segmentation fault" message. I submitted a bug report (bug #16490) and very soon some good people prepared a patch for both gcc 3.4.1 and 3.5. The patch works well, but it doesn't solve the problem which I already discovered last year. The problem is the following: once again the first compilation pass is alright, and also the profiling information is generated, but the second pass finishes with the following message: "error: coverage mismatch (...)" (gcc 3.4.1) or "error: corrupted profile info (...)" (gcc 3.5). Well, the good news is that it crashes only for a few source files, so the problem can be solved by simply deleting the profile files that cause the errors. For gcc 3.4.1 these will be:

- `graf/src/TGraphAsymmErrors.gc*`,
- `treeplayer/src/TTreeProxyGenerator.gc*`.

Unfortunately, compiling ROOT with gcc 3.5 is more complicated. The latest snapshot at the moment (2004-08-08), always fails with "segmentation fault", even if no optimization is used – I've submitted this bug some time ago, but there is still no solution available (bug #17183). Fortunately, the snapshot 2004-07-04, patched against the bug #16490 is much more stable (although one has to keep in mind that gcc 3.5 is still marked as experimental!). It works well with standard optimizations, but fails when PGO is to be used. The solution is the following. Firstly, one has to delete all the corrupted profile files (before the second compilation pass), namely:

- base/src/TFile.gc*,
- base/src/TRef.gc*,
- matrix/src/TDecompQRH.gc*,
- graf/src/TGraphAsymmErrors.gc*,
- histpainter/src/THistPainter.gc*,
- treeplayer/src/TTreeProxyGenerator.gc*,
- geom/src/TGeoManager.gc*.

Next, the second compilation pass can be started. It will fail while compiling the file "`hist/src/TFormula.cxx`" (with "segmentation fault" – I haven't had time to submit this bug yet). Therefore, one has to compile this file manually with all the optimization options, except "`-fprofile-use`". After this `make` may be run once again and the second compilation pass should now finish without any more problems.

2) Intel icc 8.0

As opposite to gcc, the Intel compiler was compiling ROOT without any problems, with or without PGO or other advanced optimizations. However, in all cases when both IPO and PGO were used one benchmark program – `stressgeom` – failed while doing some tests. This means that in some cases IPO + PGO can result in invalid code being generated. This serious bug has already been reported to Intel and hopefully an appropriate patch will be available soon. Until then, we have to reject all the results for `stressgeom` and IPO + PGO.

3.3 The Results

I've made quite a lot of compilations, but it would be too much to put here the results of all of them. Instead, in Table 1 you can find all the best results for both compiler families – gcc and icc – and the corresponding optimizations. The abbreviations used: "O2" and "O3" correspond to "`-O2`" and "`-O3`" options respectively, "fm" means "`-ffast-math`", "idml" – "`-minline-int/float-divide-min-latency`", "ismt" – "`-minline-sqrt-min-throughput`", "PGO" – "profile-guided optimization", "no-prefetch" – "`-mP2OPT_hlo_prefetch=false`" (undocumented option), "no-unroll" – "`-mP2OPT_hlo_loop_unroll=false`" (undocumented option), "ftz" – "`-ftz`" and "aa" – "`-ansi_alias`" (see the previous section for description of these options). In addition to that, in Figure 1 one can observe, what is a speedup of all the benchmark programs (so of the whole package in general) when subsequent compilation options are added to all the previous ones (so, for example, "PGO" means really "PGO + all other possible optimizations" and "aa" means "standard + ipo + aa"). The symbols in the legend have the same meaning as the ones used within the table. The "standard" options are "`-O3`" for gcc and "`-O2 -ftz`" for icc.

After analyzing all the results produced by different compilers, compilation options and even different ROOT versions, I've made the following observations:

1) The Intel compiler can produce code that is about 22% faster (on average) than the one compiled by GNU gcc. This varies for different benchmarks from 14% (`bench`) up to 37% (`stressLinear`). In any case this is a significant difference.
2) Profile-guided optimization can be very fruitful on Itanium and it seems that for production applications the effect is just worth the effort. For gcc 3.4.1 the speedup is only 3% on average (from 1% for `stressLinear` to 5% for `stress`), but for gcc 3.5 it's already 6% (from 5% for `stress` and `bench` to 10% for `stressLinear`) and for icc – 9% (from 2% for `stressLinear` to 13% for `stress`).
3) The "`-ansi_alias`" option gives little on average – about 2% – but can pay off for some benchmarks, like `stressLinear` for which the speedup is around 7%. Nevertheless, one has to keep in mind that this option can be dangerous in some cases, so it should be rather used by people who know the compiled code more or less (better more).
4) Multi-file inter-procedural optimization, which unfortunately exists only in icc compilers, is really powerful stuff, especially when used together with PGO. When added to standard options, it speeds up the ROOT benchmarks by almost 5% on average (from 1% for `stressgeom` to 7% for `stress`). However, when added to PGO flags, it makes the PGO compilation faster by more than 14% (from 7% for `stressLinear` to 19% for `bench`). The only drawback of IPO is that it takes so much time – linking of main ROOT libraries (when IPO starts its job) can last for hours. And of course one has to keep in mind that icc 8.0 has still a bug that can result in invalid code when both IPO and PGO are used.
5) There is one strange thing when moving from 1.5 GHz machine to a 1.6 GHz one. The speedup should be, theoretically, around 6.7%. However, for ROOT benchmarks it was on average around 5.7% for gcc and 4.3% for icc (5% with PGO and 3.6% without PGO). This might be due to the fact that the 1.6 GHz processors used in the tests had smaller cache, but I have no prove for that.
6) Switching from ROOT v4.00.06a to v4.00.08a has brought one huge drawback – `bench` has slowed down by about 20%. I haven't found the reason, but it has to be due to recent changes in the internal structure of ROOT. The profiles don't show any obvious candidates. However, the output from the benchmark says that what is really longer in the new version of `bench` is time to read and write "vector<THit*>" – and this creates the whole difference. Now it would be good to trace, how this corresponds to the source code, but I had no time for that.

Table 1. The best results for all compilers and both machines (1.5 GHz and 1.6 GHz)

Benchmark	Best results for gcc			Best results for icc		
	1.6 GHz	1.5 GHz	Version, options	1.6 GHz	1.5 GHz	Options
stress	911	861	3.4.1, O3, fm, idml, PGO	1115	1064	O3, no-prefetch, ftz, ipo, aa, PGO
bench	758	713	3.5, O3, fm, idml, ismt, PGO	868	816	O3, ftz, ipo, aa, PGO
stressgeom	914	864	3.5, O3, fm, idml, ismt, PGO	1107	1034	O2, ftz, aa, PGO
stressLinear	574	538	3.5, O3, fm, idml, ismt, PGO	785	737	O2, ftz, ipo, aa, PGO

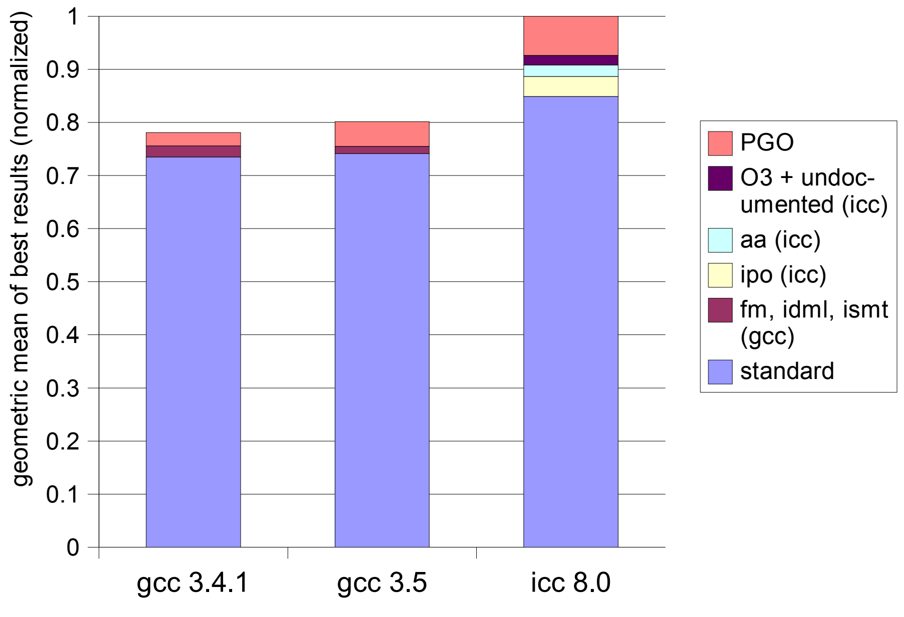

Figure 1. Summary comparison of different compilers and optimizations. Given here is a geometric mean of best results within a given criteria (normalized)

4 Profiles of the Benchmarks

Both compilers – gcc and icc – can instrument compiled code so when it's run it saves to disk some information about which instructions/functions have been called how many times and how long it took to execute them. This information can be used either by some automatic tools (e.g. for the compilers themselves when profile-guided optimization is used) or can be converted to a human-readable form, which makes efficiency analysis of a given program much easier. The problem is that the code has to be specially prepared (instrumented) for this purpose, which is not very convenient.

Fortunately, HP has recently developed a nice tool for Itanium. It runs in background and collects information about all processes running on a given machine. This information, semantically equivalent to the one that can be produced by instrumented executables, can be used for creating profiles of all the programs running on the system. They might be sometimes less accurate that the ones generated by instrumented executables, due to the sampling-related errors, but this should not be a problem as one usually needs only some general overview of what is going on inside a given program. This tool is called "`q-syscollect`" and it comes together with "`q-view`" that is responsible for converting information produced by the former to a human-readable form. The usage can be, for example, the following:

```
q-syscollect -t 300    (run q-syscollect for 300 seconds)
aProgramToBeTraced     (one or many programs one is interested in)
and after q-syscollect finishes:
q-view .q/aProgramToBeTraced-pid1234-cpu0.info#0 > profile.txt
```

Obviously, the last filename in the "`.q`" directory may be slightly different. It's worth noting that `q-syscollect` writes a separate file for each processor. Therefore, on a multi-cpu machine it might be desirable to run the traced program with the `taskset` tool, what will avoid creating multiple profiles that will have to be merged somehow (manually) later on.

The profiles of the ROOT benchmarks for icc and gcc compilers are put in Section 2 in Part IV. Here is a short summary of the hot spots of the benchmarks and of possible ways of improving their speed:

– What takes more than 20% of time in `stress` and more than 7% in `bench` are the ZIP compression and decompression functions. One should really think hard if compressing the output files is really needed (this should come with an answer to a question, whether storage is cheaper than cpu time or vice versa). If the compression is desired, one should think of optimizing the ZIP code, even if it ends up on rewriting some small functions in assembly (a good candidate is "`R__longest_match`").

– The `stressgeom` benchmark has got even a better candidate for hand-optimizing – the "`Trandom3::Rndm(int)`" method, which consumes from 25% (icc) to 32% (gcc) of the overall run time. What is really nice is that this method is particularly small, what makes it a perfect place for various experiments. To make it even simpler, I've created a small "sandbox" that contains the "`TRandom3`" class, all the header files that it needs and a little benchmark which only measures the efficiency of the "`Rndm(int)`" method[3]. I didn't have enough time to play with it, but if anybody wants to optimize this code, this is a good place to start from.

– The hot spots of the `stressLinear` benchmarks are matrix and vector operations (from `libMatrix.so`), which take more than 80% of the overall time altogether. Most of the very time consuming functions can be optimized well by a compiler itself, without any hand optimizing. Some of them already are (especially by icc), others are being looked closer by the Intel team and hopefully future Intel compilers will make them run much faster. In these circumstances it may be not very reasonable to try to optimize these functions manually on assembly level, especially because it is quite a significant amount of C++ code. On the other hand, putting a few "pragmas" in the code in order to give some hints to a compiler can be beneficial. But waiting for a new version of icc may be also a wise movement in this particular case.

5 Summary

Looking at the results of ROOT benchmarks one may ask if there is really a point in fighting so much and waiting ages for compilation to finish just to have a 20% speedup. And one may doubt even more if changing their code to make it compile well with the Intel compiler is worth the effort. The hand optimization of ROOT itself is at least as much questionable – 20%, maybe 40% in a very optimistic scenario – but it's weeks or months of difficult manual work before one can reach it. Is that all worth the effort? The answer is not as obvious as it might seem to be. Of course, when we consider only a single user, developing his or her own private application within the ROOT framework, which is going to be run for a couple of nights on a home PC, we can definitely say that this whole trouble with PGO, IPO, tricks and hand optimizing is just unreasonable. But when we switch to thinking about the LHC experiment, in which each percent of speedup is substantial money saved, then it becomes visible that one should definitely do everything to make the applications run as fast as possible. I don't mean that the advanced optimization should be used on every step of development process. It would be actually quite inconvenient. But the production versions of LHC programs should be tuned to the limit (if there is any). Not doing it will be just an irrational wasting of resources (read: money).

On the other hand, advanced optimization techniques should be used with care. It's not only because they can result in invalid code (although this is a critical issue), but also because some of them, no matter how great they are, can decrease the efficiency of

3 It can be found at: /afs/cern.ch/user/m/mkapalka/public/Trandom3benchmark.tgz

produced programs in some circumstances. It's always good to make a few compilations and compare. The same applies to hand optimizing ROOT code – not all obvious optimizations can be beneficial, as one's best ideas can be just wrongly interpreted by compilers.

Part II

Porting LHCb Software to Itanium

1 Introduction

Both C and C++, and also Fortran, are standardized. This means that ideally porting some software written in any of these languages to another platform should be as easy as recompiling it with an appropriate compiler. Unfortunately, it is usually not the case, mainly because of two reasons. Firstly, compilers tend to have some vendor-specific extensions and, by the way, may also not fulfill the standards in 100%. But one may just not use these compiler-specific constructs and avoid all the things that are either out of the standards or not handled properly by the most popular compilers. Secondly, programmers tend to make too many wrong assumptions in their code and use some "dirty" tricks which can speed up the execution of the program on one particular platform, but at the same time can make the code non-portable and, by the way, a bit more than unreadable (not to mention – difficult to debug).

I tried to port a part of the LHCb software to Itanium platform. It was not an easy task and probably without help of people like Vladimir Litvin, who had already tried porting SEAL and POOL to IA64 some time ago, and Markus Frank, one of the developers of the LHCb software, I wouldn't do so much in these 2 months. Nevertheless, the aim of this document is not only to show, how many things one has to do to compile the packages on Itanium, but also to tell the following two messages to programmers: firstly, that porting software to IA64 is not a "mission impossible", secondly, that making 64-bit compliant programs requires usually only remembering a few basic rules (and applying them, of course, in everyday work).

The following sections give a summary of what have been achieved and what does not work yet. They also point out the most common issues and problems with porting programs to the Itanium platform. However, this is only a summary. For a detailed instructions, which describe step-by-step how to compile all the packages, please refer to Section 3 in Part IV.

2 External Packages

Most of the external packages that are needed by the LHCb software compile well without any additional effort. Many of them provide some automatic tests and this is really a good thing as then it is very easy to check if the compilation is really successful. Otherwise, one can only hope that the generated executables are valid.

The following packages have been compiled without any tricks and have been proved to work well on Itanium (i.e. all of their built-in tests succeeded): bz2lib, cppunit, cmake, pcre, qmtest, uuid, zlib, MySQL, Xerces-C, expat, Grace. The EDG client libraries have been already well ported to IA64 by Andreas Unterkircher, so I could just use the binaries. Some packages, namely Boost, UnixODBC and wxPython, don't have any easy-to-use internal tests, so I couldn't check if they are really working. However, they compiled without any problems. There were also some things that didn't require compilation at all: AIDA (it's just a bunch of header files), otl4 (it's just a single header file) and Oval (it's a tool used only for testing purposes, at compilation time, so I just used the IA32 version without any problems).

However, the following packages caused some problems. They either didn't want to compile on IA64 or some of their tests failed. Fortunately, most of the problems have already been solved, but I guess there can be some hidden bugs that may come out later. These packages are:

- GSL – compiles, but one test from `cdf` subdirectory fails,
- Python – compiles, but `test_compile` fails with the message: "eval('0xffffffff') gave 4294967295, but expected -1" (this is obviously an IA64 issue, but shouldn't be dangerous – maybe the bug is in the test program itself),
- MyODBC – one test fails: `my_tran` (we can only hope that transactions won't be needed in LHCb; otherwise, somebody will have to come back to this package),
- gccxml – some manual patching was needed to make the program run correctly on more complicated source files,
- MySQL++ – fortunately, somebody has prepared an IA64 version of this package, but even this one required some manual patching and caused some more problems later; still don't know if it works correctly (there are no tests in the package, only some examples),
- Swig – a small correction to its Makefile was needed, but after that it compiled well and all the tests were ok,
- Anaphe – this requires some tricks and a lot of patience to compile: two configuration files have to be modified, some directories and symbolic links have to be created manually and a few environmental variables have to be set before it starts working; but still not all of its sub-packages want to compile (fortunately, I've managed to build all the required ones); still not tested.

3 Internal Dependencies

3.1 Things already Ported to IA64

The following applications didn't cause any major problems while compiling on IA64:

- ROOT – this is already well ported to Itanium,
- CLHEP – it required only adding the "`-fpic`" option to its `Makefile`, but besides that it's proved to work well on IA64,

- `libshift` (a part of the CASTOR project) – is not only ported to Itanium, but also the appropriate binaries can be downloaded from the project's web page,
- CERNLIB – a small modification of its configuration files is needed to make it use the "`-fpic`" compilation flag. It has been ported to IA64 by two people, but their changes, as far as I know, haven't been officially applied to the official release. However, I didn't experience any problems with the library, though it may be necessary to look at it closer in the future.

3.2 SEAL (1.3.4)

Compiling SEAL is pretty easy – only one source file has to be slightly modified to avoid some compilation errors. However, quite significant patching of the code is needed to make at least some of the tests run correctly. The majority of issues comes from two common programmers' mistakes:

- Using variables of type "`unsigned int`" instead of "`size_t`" (which is in fact "`unsigned long`") to hold results of string operations. These can return a value equal to "`std::string::npos`" that won't fit in any 32-bit variable on IA64, as it's equal to "`(unsigned long)-1`".
- Passing values of type "`int*`" to functions that expect "`long*`" instead. This results in a very difficult to trace memory corruption errors.
- Defining types that are expected to be 4-byte long integers as "`unsigned long`", which has length of 8 bytes on IA64.

I've managed to find and correct many such bugs and therefore most of the SEAL tests work now. On the other hand, quite a few tests fail on both IA32 and IA64, which means that they are not very well maintained. And this leads to an important question about how much we can trust these tests and about how large part of the SEAL code and its functionality they actually cover. But this has to be answered by SEAL developers and as it's not an IA64-specific issue, I won't bother with that.

Nevertheless, a few SEAL tests, which work well on IA32, either fail on Itanium or produce results which do not correspond well enough to the ones produced by the 32-bit code. As I wanted to check most of the LHCb dependencies in the 2 months of my stay at CERN, I haven't managed to solve these problems.

3.3 POOL (1.6.3)

Compiling POOL was much more complicated. Firstly, as this version is already pretty outdated, there were a few bugs which made it not compile with the version of GNU gcc that was used (3.2.3). Fortunately, there are already patches available at the POOL web site, as these problems are not IA64-specific. Surprisingly, SEAL also had to be patched as it had a bug that made POOL compilation fail at some point. Some other issues which were not related to the 64-bit architecture were caused by MySQL++, which, in my opinion, seems to be quite an immature project, with not very good support and not very well maintained code (fortunately, quite a few people have prepared patches which solve some problems, related either to new compilers or 64-bit architectures). Nevertheless, the most difficult thing was to find IA64-specific bugs, especially because there are quite a lot of them and they cause many different effects, from simple "segmentation faults" up to some nasty and difficult to trace memory corruption problems. Most of the issues were related to the following common programmers' mistakes:

- Using "`long`" or "`unsigned long`" (instead of "`int`" and "`unsigned int`") in places where it is not necessary, not motivated or even not reasonable. I just wonder why people use these types on 32-bit architectures if they don't expect their code to be run on 64-bit machines. On IA32 "`long int`" is equal to "`int`" and using the former should be really avoided, unless it's deeply motivated (e.g. when a given variable is supposed to keep larger numbers on 64-bit architectures – but then the whole code has to be portable!).
- Confusing "`unsigned int`" and "`size_t`". Although these types are equivalent on IA32, they are different on all 64-bit platforms. But this has already been said in the section about SEAL...
- Assuming that a given class has a specified size. This is a wrong assumption, especially when the class contains pointers or fields of type "`long int`". Not to mention things like padding (caused by data alignment requirements), which are extremely platform-specific.
- Using constants like "0xffffffff" to initialize variables of type "`long int`" (directly or indirectly) and expect later these variables to have some specified negative values (-1 in this case). Of course "`(long int)0xffffffff`" will be -1 on IA32, but not on IA64.
- Using some non-portable tricks, like very low-level bit operations or copying 4 subsequent "`char`" class fields to another class by copying 2 "`long ints`" (with some pointer casting).

Correcting bugs like these took me and Markus Frank a few days. And it's still quite a lot of things that don't work. What is more, a few things need to be rethought and then changed consequently from beginning to end. And all of this should be done as soon as possible, because when newer versions of POOL appear it will be even more difficult to apply all the patches that are already prepared, not to mention finding all the remaining bugs.

Fortunately, many tests work already and some behave only slightly different than their 32-bit counterparts (e.g. for some of them the size of generated output files is a little larger or smaller – but these files are actually created by ROOT and this framework is supposed to produce portable files on many architectures). Therefore, I believe that full porting of POOL to IA64 is feasible and can be done in the near future.

3.4 PI

The compilation problems with PI were all related to some small bugs either in its configuration files or in some external packages (e.g. missing "`-fpic`" option for some libraries). They all have been solved, except for some problems with test programs, which are not that important to waste too much time on them (but probably they can be solved quite easily).

What is more, all the tests that run well on IA32 succeed also on IA64, so there are almost no portability issues in this package. However, some output numbers (e.g. these produced by Minuit library) are slightly different and I would suggest that someone competent judge if these differences are tolerable/desirable.

4 Gaudi Framework

The problem with SEAL, POOL, PI and also with Gaudi (as well as with other LHCb software) is that they use some special and not very popular building tools. Well, they don't even use a single tool. The first three packages need "`scram`" and the rest of the LHCb software uses a completely different build environment called "`cmt`". Personally, I would really prefer typing the famous "`configure; make; make install`" than learning how to use `scram` and `cmt`, with all of their specific environmental variables, commands and, what's most horrible, own philosophy. Nevertheless, I believe that this has some explanation and once one learns how to use these tools, his or her life becomes really much easier. Thus, I won't be complaining about them anymore.

Setting up the environment for Gaudi is quite easy, although one has to use the `tcsh` shell as `bash`-compatible scripts seem to be a bit outdated. What is also quite annoying is that all the external packages, as well as all the internal dependencies, have to be put into a pre-configured directory structure that is used by all the LHCb software. I guess there must be a way to change the configuration files so that this is not necessary, but I just found it easier to copy a dozen of directories around than to find out how `cmt` can be customized.

As POOL, which is used directly by Gaudi, has changed in the porting process, Gaudi also had to be modified. Most of the changes were related to the fact that types of some of the parameters or return values of POOL methods were converted from "`(unsigned) long(*)`" to "`(unsigned) int(*)`". Therefore, the prototypes used by Gaudi didn't match any POOL functions and so the compilation failed. Nevertheless, as these were compilation problems, they were quite easy to repair, although it required changing many files.

Finally, Gaudi has been compiled together with its example programs. They have shown that the core functionality of the framework works well, but there are still problems when either IO or more advanced functionality is used. All in all, 5 examples work and 6 fail, which is not a very impressive result but also not very bad, taking into account that POOL is still not fully ported to IA64, nor SEAL. Therefore, there is still quite a lot of debugging and patching to be done, but at least voices of the prophets, who say that the LHCb-related software is completely unportable, should now calm down a bit, giving more motivation to the people who will move the thing forward.

5 Porting Software to IA64 – Comments

If programmers were more careful when writing their code, there would be no issue called "porting software to Itanium". That is because C and C++, not to mention Fortran, are really the same on both IA32 and IA64. Of course, there might be some few problems when very low level mechanisms are used, which really rely on the word size or data alignment. However, it's hardly ever the case and usually one can avoid such situations easily. Obviously, very advanced optimizations can be very platform-specific, but on the other hand it should be always possible to turn them off. What is more, it is usually better to leave all the non-safe optimizations to the last stage of software development process – when code is stable, functional and all the hot spots are identified (i.e. appropriate profiles are prepared). That has one more obvious advantage – it helps in avoiding loosing time for optimizing functions which finally consume less than a percent of run-time of all real applications.

Of course, there is a question of what to do with existing code that has to be ported to IA64. Although one can think of quite a lot of Itanium-specific issues that the software should be checked against, the experience with porting LHCb software is that there are really only a few types of them that cause most of the portability problems. What's nice is that majority of them can be discovered with use of some simple text-searching tools, like "`grep`". What's much worse is that a few ones won't issue any warnings at compilation time, no matter how advanced the compiler is. The list of the most common portability issues I had to deal with, together with some advices I've found relevant and maybe useful, follows.

1. The key thing to remember is the difference between "`int`" and "`long int`" on IA64. If a given code works well on IA32, it can work equally well without using any variables of the latter type. These two are just equivalent on 32-bit architectures and mixing both of them is just asking for portability problems, unless it's really well motivated and all the conversions in both ways are proved to be safe. One should always think twice before using "`long int`". But, of course, porting software to 64-bit platforms makes little sense when 64-bit capabilities are not used. However, it doesn't mean that all the loop counts and other variables have to become 64-bit integers, what will happen if all of them are declared as "`long int`". Nevertheless, my point here is not that one should never use "`long int`", but that one should just never assume that these two types mentioned here are equal. And doing a simple "`grep 'long int'`" is a good start of any IA64 porting process.
2. The next common issue is the difference between "`size_t`" and "`unsigned`

int". And although it's so similar to the previous point, I put it here separately to say one more thing: if a function requires an argument of type "A", feed it with a value of the type, if it returns a value of type "B", store the value in a variable of the type. It sounds so obvious, but it's not, as still so many people use "`unsigned int`" or even "`int`" in places where "`size_t`" would be a straightforward choice. Assuming that "`size_t`" is an integer type is a safe bet (the same applies to all similar types, like "`socklen_t`"). But its size is not standardized and so every conversion from "`size_t`" to other type or vice versa should be considered with care. As far as porting existing software to IA64 is concerned, doing "`grep size_t`" on the whole source code is also a good starting point.

3. Constants is another porting issue. There are two related problems. Firstly, if a constant "`unsigned int A`" is assigned a value "0xffffffff", it means only that it has a value of 0xffffffff, not -1, nor "`~0x0`". It's so obvious, but one can find so many examples of code where such wrong assumptions appear. Secondly, if a value of a constant can be computed at compile time, instead of being hard-coded in the source code, it should. For example, if a constant "B" is related to a size of a given class, it should be a function of this size, rather than a fixed value. Another thing worth mentioning is that usage of number constants should be avoided. For example, if a function returns "-1" on error, a constant like "ERROR" should be defined (and equal to -1) and the function should be documented as returning ERROR, not -1 in, these situations. This will prevent programmers from assuming that this constant has some special value which can be used later for other purposes. Coming back to the topic of porting software to IA64, "`grep -i 0xffff`" is also usually a good problems-discovery tool.

The issues mentioned here may seem to be either very obvious or very non-realistic. Therefore, I feel obliged to show a few examples of real code in which some of them appear. I won't give any comments, as this code should explain everything. For more examples, please refer to Section 3 in Part IV and the patches for SEAL, POOL, PI and Gaudi.

Example #1

Here "`lnk.second`" is of type "`unsigned long`" and can have a special initial value of "0xffffffff", for which the following loop should be executed "`cnt_size` + 1" times.

```
const size_t cnt_size = nextRecordId()-stk_size;
for(size_t j=long(lnk.second); long(j) < long(cnt_size); ++j) {
   ...
}
```

Example #2
Note that "`unsigned long`" is 4-byte long on IA32 and 8-byte long on IA64.

```
class X {
  public:
  ...
  unsigned char Data4[8];
  ...
  /// Assignment operator
  X& operator=(const X& g) {
    unsigned long       *p = (unsigned long*)&Data4[0];
    const unsigned long *q = (const unsigned long*)&g.Data4[0];
    *(p+1) = *(q+1);
    *p     = *q;
  }
```

Example #3
Note that "`std::string::npos`" constant won't fit into "`unsigned int`" on IA64.

```
std::string dName = "~" + name();
unsigned pos = dName.find("<");
if (pos != std::string::npos) {
... // this will be always executed (!)
}
```

6 Summary

To sum the things up, a substantial part of the LHCb software has been already ported to IA64, either by myself or by many other people. There is still at least as much work left to be done, but a few good lessons have already been learned. I think that now not only it will be easier to proceed with the remaining issues, but also the LHCb code being developed at the moment will be much more 64-bit aware.

Part III
Improving GNU Compilers

1 Motivation

Code produced by GNU compilers can be even 20% slower than corresponding code generated by Intel ones. This doesn't make much difference on home computers, as they are already so fast that an average user just won't feel the speedup. However, when such demanding experiments as LHC are concerned, every percent counts, as it corresponds to substantial amount of money.

The choice should be simple: if icc is better than gcc, let's use it and not bother with its free competitor. The problem is, however, that currently not all applications can be compiled with Intel icc. They can be made icc-compatible, but this may require some effort – either with adjusting configuration files, or maybe even with changing their source code. What's more, it would be better to have an option and, obviously, a good one would be gcc as this is the compiler which has been used by scientists for years and it's just what they got used to. Therefore, one should think of many possibilities: not only of "porting" applications to icc, but also of improving gcc on Itanium.

2 Possibilities

Physics or, in general, scientific applications do quite a lot of advanced mathematical computations. Therefore, the speed of basic mathematical functions, implemented either in software or in hardware, is essential. Itanium processors have only simple operations implemented on the chip and so more complicated computations, like division, square root or trigonometrical functions, have to be converted to sequences of simpler instructions by a compiler. Therefore, how well the compiler do this conversion will significantly influence the performance of the generated code.

The problem with GNU compilers is that they generate code in which most of advanced mathematical operations are implemented as calls to appropriate library functions. This has two drawbacks. Firstly, there is some overhead related to procedure calling (passing parameters, saving and restoring registers, jumps to and from the function) which can be high as compared to the time of executing a single mathematical operation. Secondly, this way the compiler cannot make a few mathematical functions be executed in parallel, what may result in not using given processor units in 100%.

Fortunately, the newer versions of gcc already allow for making some basic mathematical operations (division and, on gcc 3.5, square root) inlined on Itanium. Integer or floating point divisions can be expanded as latency- or throughput-optimized sequences of instructions. However, square root still doesn't have its latency-optimized version and so it would be desirable to implement it. It would be also very good to make other frequently-used functions inlined, as this may improve performance of compiled scientific applications.

A good place to start improving gcc at is the file `gcc/config/ia64/ia64.md`. It's written in the RTL language (see gcc resources). It contains the description of the IA64 platform – what kinds of operations are supported, what arguments they can take, how more complicated functions can be decomposed into simpler ones, which peephole optimizations can be performed and so on. Together with a few C source files and a configuration header file, all placed in the `gcc/config/ia64` directory, it makes a complete backend for Itanium platform. Therefore, to implement a new inlined function, one would usually only need to add an appropriate RTL description in the `ia64.md` file. Unfortunately, as shown in the next section, it's not that easy.

3 Problems

I started my experiments with trying to construct a latency-optimized inlined square root. As the throughput-optimized version is already implemented in gcc 3.5, it seemed to me that it's quite a feasible task. However, the first problem that appeared was that there is not much of any good documentation about hot to deal with RTL code. What is available are some technical and difficult to understand documents and also some very platform-specific descriptions of concrete porting experiments. One has to really get deep into gcc internals and its concepts to be able to do anything reasonable with RTL files. Well, one cannot expect a tutorial to be written for these few people who will ever try to port gcc to some other architectures.

There is also a difficulty in making an inlined latency-optimized square root, as it requires 6 constants to be put in registers before its code is executed (as compared to one simple constant needed by its throughput-optimized counterpart). This raises a question, whether a few subsequent square root operations (e.g. in a loop) can use a common constant-initialization block which will be executed only once, before all of them. If not, making the latency-optimized square root is just unreasonable, as with all this initialization overhead it can be slower than the throughput-optimized one. And for me it's still an open question, because I didn't have time and motivation (see below) to make more investigations in this area.

Nevertheless, the major problem is not the documentation, nor the constants. Making functions inlined can be really beneficial on IA64, but only when a few of them can be executed in parallel. Both division and square root operations consist of a few of "multiply and add" operations, which take a few cycles to execute. Therefore, after each of them is started there is a gap in which some units of Itanium are free and can be used for some other purposes. For example, if there is enough of CPU resources (like in Itanium 2 processors), two (or even more) "multiply and add" operations can overlap, and so two (at least) division or square root operations can be done in parallel. Obviously, this will require assigning different “scratch” registers to them, so they don't conflict. Unfortunately, in GNU compilers allocation of registers precedes converting mathematical operations to their inlined (final) form. Therefore, subsequent divisions or square roots use the same registers for their computations and so they cannot overlap. They will be put serially by the scheduler and so they won't have any chance to be executed in parallel. And this is a serious problem, as making functions inlined on Itanium makes little sense when they are going to be executed one by one anyway. Of course, doing it one removes procedure call overhead, but this is actually relatively small on IA64 as the architecture has a few very nice features that can speed up the process significantly (e.g. register stack frames).

There is an ongoing discussion on how the things can be improved in gcc. One conclusion now is that it won't be that easy. It might even require changing some of the platform-independent code of GNU compilers. And for sure it will take some time. Therefore, I stopped the experiments with gcc, focusing on more feasible tasks, described in this document. Hopefully somebody will come back to this topic later.

Part IV
Appendices

1 Scripts

Before anyone starts laughing and saying that all these scripts are so badly written, obscure, non-commented, difficult to use and so on, and before I receive hundreds of advices about how better they can be, and with how little effort, etc. I have to say a few things. Firstly, I made these scripts for myself, to ease my work, and I've put them here only in case anyone wondered how I did some things. Secondly, these scripts were evolving all the time as I was discovering new things and I really had no time to make refactoring or cleaning of them. Thirdly, they should work, but in some circumstances they can fail. They worked for me at least. Anyway, use it at your own risk. You have been warned[4].

1.1 Setting Up ROOT Environment (`rootenv`)

The script should be sourced from a ROOT directory. It sets the appropriate environmental variables. It does not need any parameters (it assumes current directory to be `ROOTSYS`).

```
export ROOTSYS=`pwd`
export PATH=$ROOTSYS/bin:$PATH
if [ $LD_LIBRARY_PATH ]
  then export LD_LIBRARY_PATH=$ROOTSYS/lib:.:$LD_LIBRARY_PATH
else
  export LD_LIBRARY_PATH=$ROOTSYS/lib:.
fi
```

1.2 Compiling ROOT (`rcomp`)

The script will compile ROOT with either gcc or icc, using single- or double-pass compilation. Run it without any parameters to see its syntax. The script will put all the files in a subdirectory named after version of ROOT used, compiler, its version and options. By default ROOT version 4.00.08a is used, but it can be changed with the environmental variable `ROOT_VERSION`. It's worth noting that there are a few aliases for the compiler options that are just too long to type them all the time. They're expanded to their full version by the script. These are:

– "`-idml`" = "`-minline-int-divide-min-latency -minline-float-`

4 The scripts are in: /afs/cern.ch/user/m/mkapalka/public/scripts.tgz

divide-min-latency”,
- “-ismt” = “-minline-sqrt-min-throughput”,
- “-no-prefetch” = “-mP2OPT_hlo_prefetch=false”,
- “-no-unroll” = “-mP2OPT_hlo_loop_unroll=false”.

```
if [ $# -le 2 ]
  then echo 'syntax: rcomp gcc|icc|icpc s|pgo [compiler options]'
  echo "gcc  -- for GNU gcc/g++ compilers"
  echo "icc  -- for Intel icc compilers"
  echo "icpc -- as above, but linking with icpc"
  echo "s    -- single-pass compilation"
  echo "pgo  -- profile-guided optimization (2-pass)"
  exit 1
fi

COMPILER=$1
RUN=$2
shift 2
ORIGOPTFLAGS=$*
ROOTSTDDIR=root

if [ "a$ROOT_VERSION" == "a" ]
  then ROOT_VERSION=4.00.08a
  ROOTSTDDIR=root-v4-00-08a
fi

COMPNAME=$COMPILER
if [ $COMPILER == icc -o $COMPILER == icpc ]
  then COMPNAME=ecc
fi
ARCH="linuxia64$COMPNAME"
ROOTVER=`echo $ORIGOPTFLAGS | tr -d ' '`

if [ $COMPILER == gcc ]
  then COMPVER=`$COMPILER --version | head -n 1 | cut -f 3 -d ' '`
  COMPVERFULL=`$COMPILER --version`
else
  COMPVER=`$COMPILER --version | tr -d ' '`
  COMPVERFULL=`$COMPILER -V 2>&1`
fi

if [ $RUN == pgo ]
  then RUNNAME="-PGO"
else
  RUNNAME=""
fi

ROOTVER="root-v${ROOT_VERSION}-${COMPILER}-${COMPVER}${RUNNAME}
${ROOTVER}"
ROOTDIR="`pwd`/$ROOTVER"
```

```
if [ $COMPILER == gcc ]
  then PGO1FLAGS="-fprofile-generate"
  PGO2FLAGS="-fprofile-use"
else
  PGO1FLAGS="-prof_gen -prof_dir=$ROOTDIR/profile"
  PGO2FLAGS="-prof_use -prof_dir=$ROOTDIR/profile -w"
fi

if [ $COMPILER == gcc ]
  then ORIGOPTFLAGS=`echo $ORIGOPTFLAGS | awk '{ gsub(/-idml/,
"-minline-int-divide-min-latency -minline-float-divide-min-
latency"); print }'`
  ORIGOPTFLAGS=`echo $ORIGOPTFLAGS | awk '{ gsub(/-ismt/,
"-minline-sqrt-max-throughput"); print }'`
else
  ORIGOPTFLAGS=`echo $ORIGOPTFLAGS | awk '{ gsub(/-no-prefetch/,
"-mP2OPT_hlo_prefetch=false"); print }'`
  ORIGOPTFLAGS=`echo $ORIGOPTFLAGS | awk '{ gsub(/-no-unroll/,
"-mP2OPT_hlo_loop_unroll=false"); print }'`
fi

tar -xzf root_v${ROOT_VERSION}.source.tar.gz
[ $? == 0 ] || exit $?
mv $ROOTSTDDIR $ROOTVER
[ $? == 0 ] || exit $?

cd $ROOTDIR
if [ $RUN == pgo ]
  then mkdir profile
  OPTFLAGS="${PGO1FLAGS} ${ORIGOPTFLAGS}"
  NP="-prof"
else
  OPTFLAGS=$ORIGOPTFLAGS
fi

echo "Compiling ROOT v${ROOT_VERSION}"
echo "Directory: $ROOTDIR"
echo "Compiler flags: $OPTFLAGS"

awk "{ if(\$0 ~ /^OPTFLAGS[ ]+=/) print \"OPTFLAGS = $OPTFLAGS\";
else print }" < config/Makefile.$ARCH > tmp
[ $? == 0 ] || exit $?
mv tmp config/Makefile.$ARCH

awk "{ if(\$0 ~ /^CXXFLAGS[ ]+=/) print \"CXXFLAGS = $OPTFLAGS
-fPIC\"; else if(\$0 ~ /^LDFLAGS[ ]+=/) print \"LDFLAGS =
$OPTFLAGS\"; else print }" < test/Makefile.arch > tmp
[ $? == 0 ] || exit $?
mv tmp test/Makefile.arch

awk '{ if($0 ~ /^NOOPT[ ]+=/) print "NOOPT = $(OPT)"; else print }'
< config/Makefile.$ARCH > tmp
```

```
[ $? == 0 ] || exit $?
mv tmp config/Makefile.$ARCH

if [ $COMPILER == icpc ]
  then awk '{ if($0 ~ /^LD[ ]+/) print "LD = icpc"; else print }' <
config/Makefile.$ARCH > tmp
  [ $? == 0 ] || exit $?
  mv tmp config/Makefile.$ARCH
fi

./configure $ARCH
make -j8 2>&1 | tee compile${NP}.out.txt
[ $? == 0 ] || exit $?

export ROOTSYS=`pwd`
export PATH=$ROOTSYS/bin:$PATH
if [ $LD_LIBRARY_PATH ]
  then export LD_LIBRARY_PATH=$ROOTSYS/lib:.:$LD_LIBRARY_PATH
else
  export LD_LIBRARY_PATH=$ROOTSYS/lib:.
fi

cd test
make stress bench stressgeom stressLinear 2>&1 | tee tstcomp${NP}.
out.txt
[ $? == 0 ] || exit $?

echo stress${NP}
./stress -b -q > stress${NP}.res.txt
echo bench${NP}
./bench -b -q > bench${NP}.res.txt
echo stressgeom${NP}
./stressgeom > stressgeom${NP}.res.txt
echo stressLinear${NP}
./stressLinear > stressLinear${NP}.res.txt

cd ..

if [ $RUN == pgo ]
  then make distclean
  cd test
  make distclean
  cd ..

  OPTFLAGS="${PGO2FLAGS} ${ORIGOPTFLAGS}"

  echo $OPTFLAGS

  awk "{ if(\$0 ~ /^OPTFLAGS/) print \"OPTFLAGS = $OPTFLAGS\"; else
print }" < config/Makefile.$ARCH > tmp
  mv tmp config/Makefile.$ARCH
  awk "{ if(\$0 ~ /^CXXFLAGS[ ]+=/) print \"CXXFLAGS = $OPTFLAGS
-fPIC\"; else if(\$0 ~ /^LDFLAGS/) print \"LDFLAGS = $OPTFLAGS\";
else print }" < test/Makefile.arch > tmp
  mv tmp test/Makefile.arch

  ./configure $ARCH
  make 2>&1 | tee compile.out.txt
  [ $? == 0 ] || exit $?

  cd test
  make stress bench stressgeom stressLinear 2>&1 | tee
tstcomp.out.txt
  [ $? == 0 ] || exit $?

  echo stress
  ./stress -b -q > stress.res.txt
  echo bench
  ./bench -b -q > bench.res.txt
  echo stressgeom
  ./stressgeom > stressgeom.res.txt
  echo stressLinear
  ./stressLinear > stressLinear.res.txt

  cd ..
fi

grep FAILED test/*.res.txt

uname -a > results.txt

echo -n "stress: " >> results.txt
grep ROOTMARKS test/stress.res.txt | tr -d '=' | tr -s ' ' | cut -f
3 -d ' ' >> results.txt
echo -n "bench: " >> results.txt
grep ROOTMARKS test/bench.res.txt | tr -d '=' | tr -s ' ' | cut -f
4 -d ' ' >> results.txt
echo -n "stressgeom: " >> results.txt
grep ROOTMARKS test/stressgeom.res.txt | tr -d '=' | tr -s ' ' |
cut -f 3 -d ' ' >> results.txt
echo -n "stressLinear: " >> results.txt
grep ROOTMARKS test/stressLinear.res.txt | tr -d '=' | tr -s ' ' |
cut -f 3 -d ' ' >> results.txt

cat results.txt

echo $COMPVERFULL > compiler.txt
```

1.3 Running ROOT Benchmarks (`testroot`)

The script should be called from the root directory of the ROOT compilation to be tested. It runs each benchmark 4 times, each time on a single cpu (cpu 0, then cpu 1, then cpu 0 and finally cpu 1 – on a dual-cpu machine) and computes the average of the 4 numbers. It puts all the results and outputs of the benchmark programs in `results.final.txt` and `*-final.res.txt` files respectively.

```
export ROOTSYS=`pwd`
export PATH=$ROOTSYS/bin:$PATH
if [ $LD_LIBRARY_PATH ]
  then export LD_LIBRARY_PATH=$ROOTSYS/lib:.:$LD_LIBRARY_PATH
else
  export LD_LIBRARY_PATH=$ROOTSYS/lib:.
fi

cd test
echo stress
taskset 01 -- ./stress -b -q > stress-final.res.txt
taskset 02 -- ./stress -b -q >> stress-final.res.txt
taskset 01 -- ./stress -b -q >> stress-final.res.txt
taskset 02 -- ./stress -b -q >> stress-final.res.txt
echo bench
taskset 01 -- ./bench -b -q > bench-final.res.txt
taskset 02 -- ./bench -b -q >> bench-final.res.txt
taskset 01 -- ./bench -b -q >> bench-final.res.txt
taskset 02 -- ./bench -b -q >> bench-final.res.txt
echo stressgeom
taskset 01 -- ./stressgeom > stressgeom-final.res.txt
taskset 02 -- ./stressgeom >> stressgeom-final.res.txt
taskset 01 -- ./stressgeom >> stressgeom-final.res.txt
taskset 02 -- ./stressgeom >> stressgeom-final.res.txt
echo stressLinear
taskset 01 -- ./stressLinear > stressLinear-final.res.txt
taskset 02 -- ./stressLinear >> stressLinear-final.res.txt
taskset 01 -- ./stressLinear >> stressLinear-final.res.txt
taskset 02 -- ./stressLinear >> stressLinear-final.res.txt

cd ..

uname -a > results-final.txt

echo -n "stress: " >> results-final.txt
s=`grep FAILED test/stress-final.res.txt`
if [ -z "$s" ] ; then echo OK >> results-final.txt ; else echo
FAILED >> results-final.txt ; fi
res=`grep ROOTMARKS test/stress-final.res.txt | tr -d '=' | tr -s '
' | cut -f 3 -d ' '`
sum='0'
for r in $res; do sum="$sum + $r"; done
avg=`echo "scale=2; ($sum)/4" | bc`
echo $res >> results-final.txt
echo "------" >> results-final.txt
echo $avg >> results-final.txt
echo -n "bench: " >> results-final.txt
s=`grep FAILED test/bench-final.res.txt`
if [ -z "$s" ] ; then echo OK >> results-final.txt ; else echo
FAILED >> results-final.txt ; fi
res=`grep ROOTMARKS test/bench-final.res.txt | tr -d '=' | tr -s '
' | cut -f 4 -d ' '`
sum='0'
for r in $res; do sum="$sum + $r"; done
avg=`echo "scale=2; ($sum)/4" | bc`
echo $res >> results-final.txt
echo "------" >> results-final.txt
echo $avg >> results-final.txt

echo -n "stressgeom: " >> results-final.txt
s=`grep FAILED test/stressgeom-final.res.txt`
if [ -z "$s" ] ; then echo OK >> results-final.txt ; else echo
FAILED >> results-final.txt ; fi
res=`grep ROOTMARKS test/stressgeom-final.res.txt | tr -d '=' | tr
-s ' ' | cut -f 3 -d ' '`
sum='0'
for r in $res; do sum="$sum + $r"; done
avg=`echo "scale=2; ($sum)/4" | bc`
echo $res >> results-final.txt
echo "------" >> results-final.txt
echo $avg >> results-final.txt

echo -n "stressLinear: " >> results-final.txt
s=`grep FAILED test/stressLinear-final.res.txt`
if [ -z "$s" ] ; then echo OK >> results-final.txt ; else echo
FAILED >> results-final.txt ; fi
res=`grep ROOTMARKS test/stressLinear-final.res.txt | tr -d '=' |
tr -s ' ' | cut -f 3 -d ' '`
sum='0'
for r in $res; do sum="$sum + $r"; done
avg=`echo "scale=2; ($sum)/4" | bc`
echo $res >> results-final.txt
echo "------" >> results-final.txt
echo $avg >> results-final.txt

cat results-final.txt
```

1.4 Copying ROOT Binaries between Nodes (`rootscp`)

It uses `ssh` to copy the ROOT files, but only the ones that are necessary to run all the benchmarks, from one node to another. It takes two parameters: host name (the source of the files) and the ROOT directory on the remote host. The copied binary files are put on local machine in an appropriate subdirectory of the current directory.

```
SRCHOST=$1
SRCDIR=$2
```

```
DESTDIR=`basename $SRCDIR`

DIRS='bin cint/include cint/lib cint/stl lib etc include fonts'
FILES='cint/MAKEINFO test/libEvent.so test/stress test/TBench.so
test/bench test/stressgeom test/stressLinear'

mkdir $DESTDIR
cd $DESTDIR
ssh $SRCHOST "cd $SRCDIR; tar -czf - $DIRS $FILES" | tar -xzf -
```

1.5 Setting up SEAL/POOL Environment

These commands will set up the build environment for SEAL and POOL (the `setenv` script):

```
export PATH=$PATH:/afs/cern.ch/sw/lcg/app/spi/scram
export SCRAM_ARCH=rh73_gcc32_dbg
```

The following script (`pool_settstenv`) will prepare environment in which POOL tests can be executed. It is probably better to use "`eval `scram runtime -sh``" for that purpose (although I haven't tried it), but this script can also be useful:

```
BDIR=/data1/mkapalka/build
SDIR=/data1/mkapalka/packages/POOL/POOL_1_6_3/rh73_gcc32_dbg

BL=$BDIR/boost:$BDIR/bzip2/lib:$BDIR/clhep/lib:$BDIR/cppunit/lib:$B
DIR/edg-rls-client/lib:$BDIR/expat/lib:$BDIR/gsl/lib:$BDIR/mysql/li
b:$BDIR/mysql++/lib:$BDIR/pcre/lib:$BDIR/pcre/lib:$BDIR/python/lib:
$BDIR/qmtest/lib:$BDIR/root/lib/root:$BDIR/uuid/lib:$BDIR/seal/rh73
_gcc32_dbg/lib:$BDIR/swig/lib:$BDIR/unixodbc/lib:$BDIR/uuid/lib:$BD
IR/wxpython/lib:$BDIR/xerces-c/lib:$BDIR/zlib/lib

SL=$SDIR/lib:$SDIR/tests/lib

export LD_LIBRARY_PATH=$BL:$SL

export PATH=$PATH:$BDIR/oval/bin:$SDIR/bin:$SDIR/tests/bin
export SEAL_PLUGINS=$SDIR/lib/modules:$SDIR/tests/lib/modules
```

2 Short Profiles of ROOT Benchmarks

2.1 GNU gcc

All the profiles have been produced with the binaries compiled by gcc 3.5 (shapshot 2004-07-04, patch against bug #16490) with the following options: "PGO + `-O3 -ffast-math -minline-int/float-divide-min-latency -minline-sqrt-`

min-throughput".

stress

```
Command: ./stress -b -q
Flat profile of CPU_CYCLES in stress-pid15221-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                                 name
 21.30      9.45      9.45     57.8k      164u      164u
                                                     R__Inflate_codes
  8.04      3.57     13.02     20.8k      171u      525u
                                                           R__Deflate
  6.25      2.77     15.79     51.4M     54.0n     54.0n
                                                     R__longest_match
  5.26      2.33     18.13      125M     18.6n     18.6n
                                                          R__ct_tally
  4.17      1.85     19.98      153M     12.1n     12.1n
                                                         R__send_bits
  3.16      1.40     21.38         -         -         -
                                                    _init<libCore.so>
  2.78      1.23     22.61     16.1M     76.6n     76.6n
                        TBuffer::ReadFastArrayDouble32(double*, int)
  2.76      1.23     23.84     14.7k     83.4u     83.4u
                                                    R__compress_block
  2.76      1.22     25.06     35.5M     34.5n     34.5n
                                                               memcpy
  2.31      1.02     26.09      892k     1.15u     3.27u
      TStreamerInfo::ReadBuffer(TBuffer&, void*, int, int, int, int)
  2.06      0.92     27.00     66.6k     13.7u     13.7u
                                                        R__build_tree
  1.53      0.68     27.68     55.2M     12.3n     12.3n
                                          TBuffer::operator>>(float&)
  1.43      0.63     28.31      146k     4.35u     4.35u
                                                        R__huft_build
  1.41      0.63     28.94     29.0M     21.5n     21.5n
                                           TBuffer::operator<<(float)
  1.38      0.61     29.55     8.01M     76.5n      138n
                       TTreeFormula::EvalInstance(int, char const**)
  1.08      0.48     30.03     10.4M     45.9n     98.6n
                                                               malloc
  1.03      0.46     30.49     7.60M     59.9n     59.9n
TAxis::FindBin(double)
  1.02      0.45     30.94     10.4M     43.7n     52.9n
                                                          _int_malloc
```

bench

```
Command: ./bench -b -q
Flat profile of CPU_CYCLES in bench-pid15226-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                                name
  7.53      1.60      1.60     19.8M     80.8n     80.8n
                                                    R__longest_match
  7.01      1.49      3.09         -         -         -
                                                          R__Deflate
  4.82      1.03      4.12     3.44k      298u      298u
                                                    R__Inflate_codes
  4.79      1.02      5.13     6.16M      165n      165n
                            TBuffer::WriteFastArray(int const*, int)
  4.19      0.89      6.02     6.26M      142n      142n
                                   TBuffer::ReadFastArray(int*, int)
  3.74      0.80      6.82     16.0M     49.8n     93.3n
                                                              malloc
  3.56      0.76      7.58     12.9M     58.6n      163n
                                        TRandom::Gaus(double, double)
  3.42      0.73      8.30     30.6M     23.7n     23.7n
                                                         R__ct_tally
  3.29      0.70      9.00         -         -         -
                                                   _init<libCore.so>
  2.99      0.64      9.64     53.0M     12.0n     12.0n
                                                        R__send_bits
  2.88      0.61     10.25     15.8M     38.6n     56.5n
                                                               cfree
  2.68      0.57     10.82     16.1M     35.3n     43.1n
                                                         _int_malloc
  2.43      0.52     11.33     26.6M     19.4n     19.4n
                                                  TRandom::Rndm(int)
  2.05      0.44     11.77     6.79k     64.1u     64.1u
                                                   R__compress_block
  1.96      0.42     12.19     13.1M     31.7n     31.7n
                                                                 cos
  1.92      0.41     12.60     13.2M     31.1n     31.1n
                                                        __ieee754_log
  1.79      0.38     12.98     9.70M     39.3n     39.3n
                                                              memcpy
  1.60      0.34     13.32     4.59M     74.1n      531n
                                                      THit::Set(int)
  1.51      0.32     13.64     1.77M      181n      849n
     TStreamerInfo::ReadBuffer(TBuffer&, void*, int, int, int, int)
  1.39      0.30     13.93     4.97M     59.5n     59.5n
                            TExMap::FindElement(unsigned long, long)
  1.34      0.29     14.22     1.66M      171n      992n
 TStreamerInfo::WriteBufferAux(TBuffer&, char**, int, int, int, int)
  1.33      0.28     14.50     15.9M     17.8n     17.8n
                                                           _int_free
  1.29      0.27     14.78     4.04M     67.8n     67.8n
                               TExMap::GetValue(unsigned long, long)
  1.12      0.24     15.01     2.38M      100n      100n
                                       TMath::Hash(void const*, int)
```

stressgeom

```
Command: ./stressgeom
Flat profile of CPU_CYCLES in stressgeom-pid15232-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                                name
 32.28      1.54      1.54     48.5M     31.7n     31.7n
                                                 TRandom3::Rndm(int)
 10.76      0.51      2.05     3.01M      171n      171n
                                   TGeoArb8::Contains(double*) const
  9.40      0.45      2.50         -         -         -
                                                  sample_volume(int)
  4.66      0.22      2.72     1.25M      178n      194n
                 TGeoVoxelFinder::GetNextCandidates(double*, int&)
  3.61      0.17      2.89     1.06M      163n      202n
                                   TGeoPgon::Contains(double*) const
  2.75      0.13      3.03     2.36M     55.4n     55.4n
                                                              __atan2
  2.56      0.12      3.15     1.98M     61.5n     61.5n
                                   TGeoCone::Contains(double*) const
  1.99      0.10      3.24     1.72M     55.4n     55.4n
                     TMath::BinarySearch(int, double const*, double)
  1.70      0.08      3.32     1.01M     79.9n      103n
                                 TGeoSphere::Contains(double*) const
  1.66      0.08      3.40     1.01M     78.4n     78.4n
                                   TGeoPcon::Contains(double*) const
  1.51      0.07      3.48     2.01M     35.9n     35.9n
                                   TGeoTube::Contains(double*) const
  1.47      0.07      3.55     1.68M     41.6n     41.6n
                                   TGeoBBox::Contains(double*) const
  1.38      0.07      3.61      997k     66.2n     66.2n
                                   TGeoEltu::Contains(double*) const
  1.36      0.07      3.68     2.00M     32.5n     32.5n
                                                                 cos
  1.28      0.06      3.74     1.03M     59.5n     59.5n
                                   TGeoTrd1::Contains(double*) const
  1.26      0.06      3.80      989k     60.7n     60.7n
                                   TGeoCtub::Contains(double*) const
  1.20      0.06      3.85     1.02M     56.0n     56.0n
                                   TGeoTrd2::Contains(double*) const
  1.15      0.06      3.91         -         -         -
                                                 _init<libGeom.so>
  1.03      0.05      3.96     2.32M     21.1n     21.1n
```

stressLinear

```
Command: ./stressLinear
Flat profile of CPU_CYCLES in stressLinear-pid15236-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                              name
 15.07      5.30      5.30      112k     47.4u     47.4u
       TMatrixDSparse::AMultBt(TMatrixDSparse const&, TMatrixDSparse
                                                      const&, int)
  8.93      3.14      8.45     9.42M      334n      334n
         ApplyHouseHolder(TVectorD const&, double, double, int, int,
                                                  TMatrixDColumn&)
  6.87      2.42     10.87      241k     10.0u     10.0u
               TMatrixDBase::DoubleLexSort(int, int*, int*, double*)
  6.41      2.26     13.12     69.7k     32.4u     32.4u
                               TMatrixD::operator*=(TMatrixD const&)
  6.28      2.21     15.34     91.7k     24.1u     24.1u
             TMatrixD::AMultB(TMatrixD const&, TMatrixD const&, int)
  4.06      1.43     16.76      195k     7.33u     11.8u
 TDecompSVD::Diag_3(TMatrixD&, TMatrixD&, TVectorD&, TVectorD&, int,
                                                              int)
  3.72      1.31     18.07     62.5M     21.0n     21.0n
                                TMatrixD::operator()(int, int) const
  3.21      1.13     19.20     94.1k     12.0u     12.0u
 TDecompLU::DecomposeLUCrout(TMatrixD&, int*, double&, double, int&)
  2.63      0.93     20.13     66.8k     13.9u     13.9u
                       TDecompLU::InvertLU(TMatrixD&, int*, double)
  2.59      0.91     21.04      380k     2.40u     2.42u
 VerifyMatrixIdentity(TMatrixDBase const&, TMatrixDBase const&, int,
                                                           double)
  2.46      0.87     21.91     4.15M      208n      208n
                  TVectorD::operator=(TMatrixDColumn_const const&)
  2.30      0.81     22.72     19.9M     40.6n     40.6n
                         TMath::BinarySearch(int, int const*, int)
  1.96      0.69     23.40     1.68M      411n      411n
         ApplyHouseHolder(TVectorD const&, double, double, int, int,
                                                     TMatrixDRow&)
  1.87      0.66     24.06     61.2M     10.7n     10.7n
                      ApplyGivens(double&, double&, double, double)
  1.75      0.62     24.68         -         -         -
                                                _init<libMatrix.so>
  1.66      0.58     25.26      184k     3.17u     3.17u
                                           MakeHilbertMat(TMatrixD&)
  1.55      0.55     25.81     19.9M     27.5n     68.0n
                          TMatrixDSparse::operator()(int, int) const
  1.45      0.51     26.32     7.41M     68.7n     71.5n
                                                       _int_malloc
  1.19       0.42      26.73       185k     2.25u     2.25u
                                           TVectorD::operator*=(TMatrixD const&)
```

2.2 Intel icc

All the profiles have been produced with the binaries compiled by icc 8.0 with the following options: “PGO + `-O2 -ftz -ipo -ansi_alias`”, except for `stressgeom`, for which IPO was not used. Unfortunately, not all the names of functions have been properly demangled – it's probably the case for all inlined functions. If in doubt what this strange names may correspond to, please refer to the gcc profiles, which are much more clear.

stress

```
Command: ./stress -b -q
Flat profile of CPU_CYCLES in stress-pid12701-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self      cumul      calls self/call  tot/call
                                                                   name
  6.34      2.23       2.23      51.2M     43.6n     43.6n
                                                       R__longest_match
  4.09      1.44       3.67          -         -         -
                                            .R__Inflate_codes__$2486_2$
  3.47      1.22       4.89          -         -         -
                                          .R__Inflate_codes__$2486_144$
  3.36      1.18       6.08          -         -         -
                                         .R__Deflate_fast$171__$2020_3$
  2.94      1.03       7.11          -         -         -
                                          .R__Inflate_codes__$2486_240$
  2.63      0.93       8.04      29.0M     32.0n     32.0n
                                                     .l__ZN7TBufferlsEf
  2.39      0.84       8.88          -         -         -
                                          .R__Inflate_codes__$2486_242$
  2.29      0.81       9.68          -         -         -
                                       .R__compress_block$171__$2034_5$
  2.14      0.75      10.44          -         -         -
                           ._ZN13TStreamerInfo10ReadBuffer__$2872_1760$
  2.12      0.75      11.18      11.4M     65.7n     65.7n
                                                ?0_memcopyA<libEvent.so>
  1.93      0.68      11.87      55.0M     12.4n     12.4n
                                             TBuffer::operator>>(float&)
  1.81      0.64      12.50          -         -         -
                                                       _init<libCore.so>
  1.68      0.59      13.09          -         -         -
                                            .R__Inflate_codes__$2486_7$
  1.51      0.53      13.63          -         -         -
                                            .R__Inflate_codes__$2486_1$
  1.45      0.51     14.14     10.0M     51.1n     51.1n
                                                       _int_malloc
  1.24      0.44     14.57     10.0M     43.3n     94.2n
                                                            malloc
  1.22      0.43     15.00     22.0M     19.4n     19.4n
                                                 memcpy<libEvent.so>
  1.19      0.42     15.42     9.72M     43.2n     68.3n
                                                             cfree
```

bench

```
Command: ./bench -b -q
Flat profile of CPU_CYCLES in bench-pid12706-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                              name
  7.18      1.34      1.34     20.0M     67.1n     67.1n
                                                  R__longest_match
  6.73      1.26      2.60     6.41M      196n      197n
                          TBuffer::WriteFastArray(int const*, int)
  3.68      0.69      3.29     16.0M     43.1n     60.6n
                                                             cfree
  3.52      0.66      3.95     16.1M     40.9n     82.6n
                                                            malloc
  3.08      0.58      4.52         -         -         -
                         ._ZN7TBuffer13ReadFastArrayEPii__$287_12$
  2.95      0.55      5.07     16.0M     34.4n     41.9n
                                                       _int_malloc
  2.31      0.43      5.51     13.2M     32.7n     32.7n
                                  ._ZN7TRandom4GausEdd__$3131_19$
  2.04      0.38      5.89         -         -         -
                                              .l__ZN7TRandom4GausEdd
  2.03      0.38      6.27     26.2M     14.5n     14.5n
                                               .l__ZN7TRandom4RndmEi
  1.82      0.34      6.61         -         -         -
                                                 _init<libCore.so>
  1.71      0.32      6.93     6.42k     49.8u     49.8u
                                                R__fill_window$171
  1.69      0.32      7.24     15.7M     20.1n     81.0n
                                                         .l__ZdlPv
  1.50      0.28      7.52         -         -         -
                                  ._ZN7TRandom4GausEdd__$3131_50$
  1.49      0.28      7.80     16.0M     17.4n     17.4n
                                                         _int_free
  1.46      0.27      8.08         -         -         -
                                   .R__Deflate_fast$171__$2020_3$
  1.25      0.23      8.31     3.28M     71.5n     71.5n
                                            ?0_memcopyA<libCore.so>
  1.15      0.22      8.53     2.29M     93.8n     93.8n
                              TExMap::Add(unsigned long, long, long)
  1.11      0.21      8.73         -         -         -
                                            TCloneshit::MakeEvent(int)
  1.03      0.19      8.93     15.8M     12.2n     95.5n
                                                             .l__Znwm
  1.03      0.19      9.12     50.6k     3.79u     3.79u
                                                    kernel:__copy_user
```

stressgeom

```
Command: ./stressgeom
Flat profile of CPU_CYCLES in stressgeom-pid14598-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                                 name
 25.45      1.03      1.03     48.7M     21.1n     21.1n
                                         _ZN8TRandom34RndmEi__hot_8_0
 11.98      0.48      1.51      27.0     17.9m     59.5m
                                                    sample_volume(int)
  3.76      0.15      1.66     1.14M      133n      176n
           $_1$_ZN15TGeoVoxelFinder17GetNextCandidatesEPdRi$TAG$GLOB
  3.00      0.12      1.79     2.42M     50.0n     50.0n
                                                               __atan2
  2.70      0.11      1.89         -         -         -
                                 ._ZNK8TGeoArb88ContainsEPd__$17_142$
  2.20      0.09      1.98         -         -         -
                                  ._ZNK8TGeoArb88ContainsEPd__$17_19$
  2.05      0.08      2.07         -         -         -
                                          ._ZN8TRandom34RndmEi__$8_9$
  1.93      0.08      2.14     1.86M     42.0n     42.0n
                                     .l__ZN5TMath12BinarySearchEiPKdd
  1.58      0.06      2.21         -         -         -
                                   ._ZNK8TGeoArb88ContainsEPd__$17_3$
  1.29      0.05      2.26         -         -         -
                                  ._ZNK8TGeoCone8ContainsEPd__$20_27$
  1.24      0.05      2.31         -         -         -
                                                          .l__Z6lengthv
  1.16      0.05      2.36      27.0     1.74m     1.74m
                                                    _init<libGeom.so>
  1.14      0.05      2.40     44.0k     1.05u     1.63u
                                                            do_lookup
  1.11      0.05      2.45     2.12M     21.3n     21.3n
                                              .l__ZN5TMath6LocMinEiPKd
  1.09      0.04      2.49         -         -         -
                                         .l__ZNK8TGeoArb88ContainsEPd
```

stressLinear

```
Command: ./stressLinear
```

```
Flat profile of CPU_CYCLES in stressLinear-pid12712-cpu1.hist#0:
 Each histogram sample counts as 1.00034m seconds

% time      self     cumul     calls self/call  tot/call
                                                       name
  8.74      2.25      2.25     14.9k      150u      156u
          TDecompSVD::Bidiagonalize(TMatrixD&, TMatrixD&, TVectorD&,
                                                     TVectorD&)
  6.99      1.80      4.04      258k     6.95u     6.95u
               TMatrixDBase::DoubleLexSort(int, int*, int*, double*)
  5.83      1.50      5.54         -         -         -
                        ._ZN14TMatrixDSparse7AMultBtERK__$138_54$
  3.94      1.01      6.55         -         -         -
                       ._ZN14TMatrixDSparse7AMultBtERK__$138_120$
  3.23      0.83      7.38         -         -         -
                           ._ZN8TMatrixD6AMultBERKS_S1_i__$78_25$
  3.18      0.82      8.20     67.9k     12.1u     12.1u
                            TMatrixD::operator*=(TMatrixD const&)
  3.16      0.81      9.01     19.7M     41.2n     41.2n
                                 .l__ZN5TMath12BinarySearchEiPKii
  2.63      0.68      9.69     3.96M      171n      171n
                       .l__ZN8TVectorDC9ERK20TMatrixDColumn_const
  2.14      0.55     10.24     7.45M     73.7n     75.0n
                                                      _int_malloc
  2.09      0.54     10.78         -         -         -
                        ._ZN14TMatrixDSparse7AMultBtERK__$138_50$
  2.05      0.53     11.31      115k     4.57u     6.84u
                               TDecompQRH::Solve(TMatrixDColumn&)
  2.02      0.52     11.82     19.7M     26.3n     67.5n
                                     .l__ZNK14TMatrixDSparseclEii
  1.96      0.50     12.33         -         -         -
                        ._ZN10TDecompSVD6Diag_3ER8TMatr__$932_56$
  1.93      0.50     12.82     39.0k     12.7u     12.7u
 TDecompQRH::QRH(TMatrixD&, TVectorD&, TVectorD&, TVectorD&, double)
  1.67      0.43     13.25         -         -         -
                       .VerifyMatrixIdentity(TM__$995_60$ const&)
  1.62      0.42     13.67         -         -         -
                       .VerifyMatrixIdentity(TM__$995_42$ const&)
  1.62      0.42     14.08         -         -         -
                       .VerifyMatrixIdentity(TM__$995_57$ const&)
  1.35      0.35     14.43         -         -         -
                       ._ZN10TDecompSVD6Diag_3ER8TMatr__$932_102$
  1.19      0.31     14.74     7.45M     41.2n      116n
                                                           malloc
  1.17      0.30     15.04         -         -         -
                          ._ZN9TDecompLU8InvertLUER8TMatr__$85_27$
  1.05      0.27     15.31         -         -         -
                        ._ZN10TDecompSVD6Diag_3ER8TMatr__$932_51$
  1.05      0.27     15.58     7.31M     36.8n     69.3n
                                                            cfree
  1.02      0.26     15.84     98.6k     2.67u     2.76u
                       .l__ZN10TDecompSVD5SolveER14TMatrixDColumn
```

3 Compiling LHCb Software on IA64 – Step by Step

In this section the following environmental variables are assumed to be defined:

– `PACKDIR` – the directory where all the external packages are being unpacked and compiled,
– `BUILDDIR` – the directory where binaries of all the external packages are to be put,
– `EXTLCG` – LCG directory with external files,
– `EXTTARFILES` – a directory where source tar files of most of the external LCG-related packages can be found.

For example, these were my settings on the machine `oplapro49`[5]:

```
export PACKDIR=/data1/mkapalka/packages
export BUILDDIR=/data1/mkapalka/build
export EXTLCG=/afs/cern.ch/sw/lcg/external
export EXTTARFILES=$EXTLCG/tarFiles
```

To simplify the description of how I changed some files to make some of the programs work, I use here the standard “diff” notation. For example the following two lines mean that in a given file file a given line containing “old line contents” has been changed to “new line contents”:

```
- old line contents
+ new line contents
```

3.1 External Packages

1) boost (1.30.2)

Some instructions are on the page: http://www.boost.org/more/getting_started.html. First one needs to compile Boost.Jam:

```
cd tools/build/jam_src
bash build.sh gcc
```

Next comes Boost. There are some non-portable declarations in file `boost/cast.hpp`, but haven't manage to change it properly.

```
export PYTHON_ROOT=/usr
tools/build/jam_src/bin.linuxia64/bjam "-sTOOLS=gcc"
copy all libs/*.a *.so --> $BUILDDIR/boost
copy directories boost and libs --> $BUILDDIR/boost
```

OK

5 The detailed directory structure of my configuration (on `oplapro49`) can be downloaded from: /afs/cern.ch/user/m/mkapalka/public/dirstruct.tgz

2) bz2lib (1.0.2)

This is quite straightforward:

```
cp $EXTTARFILES/bzip2-1.0.2.tar.gz .
make
make -f Makefile-libbz2_so
make install PREFIX=$BUILDDIR/bzip2
cp libbz2.so.1.0* $BUILDDIR/bzip2/lib/
```

All tests passed. OK

3) cppunit (1.8.0)

Very straightforward:

```
./configure --prefix=$BUILDDIR/cppunit
make
make check
make install
cp $EXTLCG/CppUnit/1.8.0/rh73_gcc323/include/CppUnit_testdriver.cpp
$BUILDDIR/cppunit/include
```

All tests passed. OK

4) cmake (2.0.2)

The tar file can be downloaded from www.cmake.org. The compilation is easy:

```
./configure --prefix=$BUILDDIR/cmake
make
make test
make install
```

All tests passed. OK

5) gccxml (0.6.0)

The version 0.4.2-patch1 of the package, which is used by LHCb software on IA32, is very buggy and I couldn't patch it enough to make it work on IA64. However, the version 0.6.0 is much more stable and didn't cause any problems later, so I strongly recommend using it[6]. Nevertheless, even this version required some manual patching – otherwise there are problems while compiling SEAL (error messages "sorry, not implemented: `fdesc_expr' not supported by dump_expr" when gccxml is used in dictionary generation process). The installation steps:

```
mkdir gccxml-0.6.0-build
cd gccxml-0.6.0-build
```

6 It can be downloaded from: http://public.kitware.com/GCC_XML/HTML/Index.html

```
cmake ../gccxml-0.6.0 -DCMAKE_INSTALL_PREFIX:PATH=$BUILDDIR/gccxml
make
make install
```

It seems to be OK.

6) gsl (1.4)

Compilation is straightforward:

```
./configure --prefix=$BUILDDIR/gsl
make
make check
make install
```

One test from `cdf` subdirectory fails, but it shouldn't be a problem. OK

7) pcre (4.4)

Well:

```
./configure --prefix=$BUILDDIR/pcre
make
make check
make install
```

All tests passed. OK

8) Python (2.2.3)

Can be downloaded from: http://www.python.org. Compilation is not very tricky:

```
./configure --prefix=$BUILDDIR/python
make OPT="-fpic -O2"
make test
mkdir .extract
cd .extract
ar xv ../libpython2.2.a
cd ..
gcc -shared -o libpython2.2.so .extract/*.o

make install
```

One test failed with the following message: “test test_compile failed -- eval('0xffffffff') gave 4294967295, but expected -1”. It's obviously an IA64 issue... But I think we can live with that (it's probably the test itself that is not IA64-compliant). OK

9) Oval (3.5.0)

It's only needed to run some tests, so one can just copy the directory – it should work like that. OK

10) qmtest (2.0.3)

The package name is `qm-2.0.3.tar.gz.` Installation:

```
./configure --prefix=$BUILDDIR/qmtest
make
make check
make install
```

All tests passed. OK

11) uuid (from e2fsprogs 1.32)

Simple. Go to the directory where uuid has been unpacked and type:

```
mkdir build
cd build
../configure --enable-elf-shlibs --prefix=$BUILDDIR/uuid
make
make check
make install
```

All tests passed. OK

12) zlib (1.1.4)

Compilation:

```
./configure --prefix=$BUILDDIR/zlib
make
make test
make install
```

All tests passed. OK

13) Valgrind (2.0.0)

This is x86 specific program, so it won't compile on IA64. And there is no IA64 port available. But it's only a debugger/memory profiler, so we won't need it (although it would be nice for debugging).

14) edg-rls-client (2.2.1)

Fortunately, Andreas Unterkircher has ported the whole EDG to IA64, so one should just get the appropriate rpms from him and unpack them. These are the following:

- `edg-gsoap-base_gcc3_2_2-1.0.8-1.ia64.rpm`
- `edg-replica-location-client-c++_gcc3_2_2-2.2.1-1.ia64.rpm`

- `edg-replica-location-index-client-c++_gcc3_2_2-2.2.0-1.ia64.rpm`
- `edg-replica-metadata-catalog-client-c++_gcc3_2_2-2.2.1-1.ia64.rpm`

I have also added the following symbolic links (their corresponding files have suffixes with version number, compiler, etc.):

- `libedg_gsoap_base.a`
- `libedg_local_replica_catalog_client.a`
- `libedg_replica_location_index_client.a`
- `libedg_replica_metadata_catalog_client.a`

OK

15) MySQL (4.0.13)

The build procedure (probably some of these steps can be skipped, but I'm not sure, which ones):

```
./configure --prefix=$BUILDDIR/mysql \
  --with-extra-charsets=complex --enable-thread-safe-client \
  --enable-local-infile
make
make install

su -
groupadd mysql
useradd -g mysql mysql
scripts/mysql_install_db

$BUILDDIR/mysql/bin/mysqld_safe
cd $BUILDDIR/mysql/sql-bench
perl run-all-tests
```

It takes a very long time to finish all tests, but all are passed. OK

16) MySQL++ (1.7.9, patched for IA64)

Compilation of this package on IA64 is a bit tricky, as this software is not well maintained (it's not a part of MySQL, but rather an unofficial its extension). There are a lot of patches for different compilers and platforms, but merging a few of them can require changing the source code by hand. I used a version patched for IA64 and I applied some more changes from other patches manually. The package can be downloaded from ftp://ftp.solnet.ch/mirror/mysql/Downloads/mysql++/mysql++-1.7.9_gcc_3.3_IA64.tar.gz. Firstly, one has to remove a bug in the `configure` script: in line 1558 the right-hand expression has to be put in double quotes. Next, one has to edit the file `sqlplusint/type_info1.hh`: line 25 has to be commented out:

```
- mysql_ti_sql_type_info (const mysql_ti_sql_type_info &b);
```

```
// can't do
+ // mysql_ti_sql_type_info (const mysql_ti_sql_type_info &b);
// can't do
```

The compilation itself is, however, straightforward:

```
./configure --prefix=$BUILDDIR/mysql++ --with-mysql=$BUILDDIR/mysql
make
make install
```

OK

17) otl4 (4.0.67)

Just unpack and copy the file `otlv4.h` to `$BUILDDIR/otl`. OK

18) UnixODBC (2.2.6)

Straightforward:

```
./configure --prefix=$BUILDDIR/unixodbc
make
make install
```

OK

19) MyODBC (3.51)

The build procedure:

```
./configure --with-mysql-path=$BUILDDIR/mysql \
  --with-unixODBC=$BUILDDIR/unixodbc \
  --prefix=$BUILDDIR/myodbc --enable-thread-safe --enable-shared
make
make install
```

To run all the tests:

```
edit odbc.ini
export ODBCINI=/patch_to_odbc.ini/odbc.ini
ln -s /var/lib/mysql/mysql.sock /tmp/mysql.sock
make test
```

One test fails: `my_tran`. I hope transactions won't be needed... Besides that, OK

20) Xerces-C (2.3.0)

Installation steps:

```
export XERCESCROOT=`pwd`
cd src/xercesc
```

```
./runConfigure -p linux -b 64 -P $BUILDDIR/xerces-c
make
make install
```

All tests passed. OK

21) wxPython (2.4.0.1)

Installation:

```
mkdir build
cd build
../configure --with-gtk --prefix=$BUILDDIR/wxpython \
  --enable-rpath=$BUILDDIR/wxpython/lib --with-opengl \
  --enable-geometry --enable-optimise --enable-debug_flag \
  --with-libjpeg=builtin --with-libpng=builtin \
  --with-libtiff=builtin --with-zlib=builtin
make
cd ../locale
make allmo
cd ../build
make install
cd ../wxPython
$BUILDDIR/python/bin/python setup.py IN_CVS_TREE=1 \
  WX_CONFIG=$BUILDDIR/wxpython/bin/wx-config build install
```

OK

22) AIDA (3.0.0)

Just unpack it – it doesn't seem like there is anything you should do with that (these are only header files). OK

23) expat (1.95.5)

Compilation is simple:

```
./configure --prefix=$BUILDDIR/expat
make
make install
```

To run tests the "check" library (8.0) is needed (see http://check.sourceforge.net). The installation of the library is fast and simple:

```
mkdir build
./configure --prefix=`pwd`/build
make
make check
make install
```

All tests for "check" are passed. Next, one can run the expat tests:

```
export LIBRARY_PATH=$PACKDIR/check-0.8.0/build/lib
export CPATH=$PACKDIR/check-0.8.0/build/include
make check
```

All tests passed. OK

24) Grace (5.1.10) – needed by Anaphe

Compilation steps:

```
./configure --prefix=$BUILDDIR/grace
make
make check
make install
```

All tests passed. OK

25) Swig (1.3.14) – needed by Anaphe

Downloaded from http://www.swig.org. Installation fails on modules “perl5” and “tcl” (no required header files or libraries on oplapro49 machine), so I removed them, as Anaphe probably needs only the Python module. However, the option “--without-tcl” doesn't seem to work, so I patched the Makefile manually: line 65:

```
- skip-tcl        = [ -z "-I/usr/include" -o -z "-L/usr/lib
-ltcl8.3${TCL_DBGX}" ]
+ skip-tcl        = [ -z "" -o -z "-L/usr/lib -ltcl8.3
${TCL_DBGX}" ]
```

The rest of the installation process is simple:

```
./configure --with-python=$BUILDDIR/python/bin/python \
  --prefix=$BUILDDIR/swig --without-perl5 --without-tcl
make
make -k check
make install
```

All tests passed. OK

26) libshift

Downloaded libshift binaries for IA64 from the CASTOR web page:
http://castor.web.cern.ch/castor/DIST/CERN/savannah/CASTOR.pkg/1.7.1.5/IA64.directory/CASTOR-client-1.7.1.5-1.longname.ia64.rpm.
Unpacked just the libraries and header files of libshift to `$BUILDDIR/libshift`. OK

3.2 Internal Dependencies

1) CLHEP (1.8.1.0)

This works pretty well:

```
cp $EXTTARFILES/clhep-1.8.1.0.tgz .
unpack & go to the directory
./configure --prefix=$BUILDDIR/clhep

edit Makefile --> add "-fpic" option to CXXFLAGS

make
make install

cd test
make
make check

cd ..
mkdir .extract
cd .extract
ar xv ../libCLHEP-1.8.1.0.a
cd ..
gcc -shared -pthread -o libCLHEP-g++.1.8.1.0.so .extract/*.o
cp libCLHEP-g++.1.8.1.0.so $BUILDDIR/clhep/lib
cd $BUILDDIR/clhep/lib
ln -s libCLHEP-g++.1.8.1.0.so libCLHEP.so
```

All tests passed. OK

2) ROOT (3.10.02)

This is quite an old version of ROOT, but compiles well on IA64 and has been already ported to this platform. The compilation is straightforward, but one has to remember to turn on the RFIO support; otherwise, PI won't compile:

```
./configure linuxia64gcc --enable-rfio \
  --with-shift-libdir=$BUILDDIR/libshift/lib \
  --with-shift-incdir=$BUILDDIR/libshift/include/shift \
  --prefix=$BUILDDIR/root
make
make install
```

OK

3) CERNLIB (2003)

I used the official version of CERNLIB. There are two people who have made ports of these libraries to IA64, but these have been officially accepted. As I didn't have any access to this "private" ports, I had to used the non-ported packages, but they seem to work, at least as far as the functionality needed by LHCb software is concerned. The only problems I've found were related to the Minuit library (a part of PACKLIB), but it's difficult to say if these are really errors, or just slightly different (better or worse) approximations.

To build CERNLIB libraries, one usually has to download all the CERNLIB `*.tar.gz` files and the `start_cern` script[7]. However, after problems with PI (relocation error) I discovered that this procedure will not, by default, compile the libraries with the required on IA64 "`-fpic`" option. I modified `linux.cf` configuration file and the `start_cern` script[8]. OK

3.3 Anaphe (5.0.6)

It needs SWIG, Grace, `libpacklib` (CERNLIB) and a lot of patience... Download the source file from http://pcitapiww.cern.ch/anaphe/download/Anaphe-5.0.6-src.tgz. Next come the tricky installation steps:

```
export \
LIBRARY_PATH=$BUILDDIR/swig/lib:$BUILDDIR/grace/grace/lib:$BUILDDIR
/expat/lib:$BUILDDIR/clhep/lib:/data1/mkapalka/libshift/lib
export \
CPATH=$BUILDDIR/grace/grace/include:$BUILDDIR/expat/include:$BUILDD
IR/clhep/include:/data1/mkapalka/libshift/include
export AIDA_DIR=$BUILDDIR/aida/cpp
export CLHEP_DIR=$BUILDDIR/clhep

admin/scripts/mkRelease.py --version 5.0.6 \
  --platf redhat73/gcc-3.2 --rel=$BUILDDIR/anaphe
```

Normally it should work, but here it will fail... Now it's time for some dirty tricks:

```
cd $BUILDDIR/anaphe/specific/redhat73/gcc-3.2
mkdir -p OtherLibs/1.0.0.1/lib
cd OtherLibs/1.0.0.1/lib
ln -s $BUILDDIR/cernlib/lib/libpacklib.a libminuit.a
ln -s /usr/lib/libg2c.so.0 libg2c-forMinuit.so

cd $BUILDDIR/anaphe/specific/redhat73/gcc-3.2
mkdir CLHEP
ln -s $BUILDDIR/clhep CLHEP/1.8.0.0
```

Now you should edit the `admin/scripts/mkRelease.py` script: comment out the two lines 454 and 455, so they look like the following:

```
#nag_inc_dir = os.environ["NAG_INCLUDE_DIR"]
```

7 See http://cernlib.web.cern.ch/cernlib/install/install.html

8 See /afs/cern.ch/user/m/mkapalka/public/cernlib2003-ia64config.tgz

```
#nag_lib_dir = os.environ["NAG_LIB_DIR"]
```

In the same file, in line 456:

```
+ optionNag = ''
- optionNag = '"USE_NAG=1" ...'
```

Next, edit the file `packages/AIDA_HBookStore/GNUmakefile` and change:

```
- CERNLIB = -static `...` -lshift
+ CERNLIB = /data1/mkapalka/build/libshift/lib/libshift.a
```

Once again:

```
admin/scripts/mkRelease.py --version 5.0.6 \
   --platf redhat73/gcc-3.2 --rel=$BUILDDIR/anaphe
```

Still some packages will fail to compile, but we will already have all what we need. OK

3.4 SEAL

1) Compiling and Testing SEAL

Firstly, SEAL configuration files have to be updated (all patches to external and internal dependencies). The files to be changed are `.SCRAM/rh73_gcc/*.dat`[9] (remark: `lib` and `include` patches in `root.dat` have to point to `ROOTSYS/lib/root` and `ROOTSYS/include/root` respectively). I guess there exists a more efficient and automatic way to do it, with use of the `scram` tool, but I didn't have enough time/patience to look for it.

Next, we can go to the right directory and set up the SEAL environment. My method was the following:

```
cd /data1/mkapalka/packages/SEAL/SEAL_1_3_4
source ../setenv
```

although I guess using "`eval `scram runtime -sh``" instead of the `setenv` script might be a better idea (I didn't know about this method at the time of porting SEAL). Once set up, we can start compiling and testing SEAL. The `scram` tool will be needed for this purpose. As it's used only at compilation phase, we don't have to port it to IA64, but we can use the x86 version instead (the `setenv` script sets the `PATH` variable appropriately). Well:

```
cd src

scram b release

or, equivalently:
```

9 See /afs/cern.ch/user/m/mkapalka/public/seal-ia64config.tgz for an example

```
scram b release-reset-arch
scram b release-build
scram b release-check
scram b release-docs (optional)
scram b release-freeze (optional -- personally I don't like it
because it makes all the files read-only...)
```

If `scram` fails with the error "/bin/sh: line 1: scram: command not found", then just copy the `scram` shell script to a directory which is added to `PATH` by default for `/bin/sh` (e.g. `~/bin` at CERN). Adding `PATH` to `.bashrc` or `.profile` won't help, as they are not read by `/bin/sh` in interactive mode.

The following paragraphs describe all the problems that I had when trying to compile and test SEAL. I have prepared a patch[10] for SEAL 1.3.4 which, after applied, should resolve all the issues mentioned below, except for a few bugs which I haven't managed to repair yet. Nevertheless, as SEAL 1.3.4 is already a bit outdated, one should consider applying the patches manually to a newest version and also trying to find more bugs (they are probably similar issues to the ones described here).

2) Compilation and Run-Time Errors

Problem: compilation error:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/IntBits.h:99: redefinition of `struct seal::IntBits<64>'
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/IntBits.h:84: previous definition of `struct
seal::IntBits<64>'
```

Remedy: changed "`#endif + #if`" to "`#elif`" in file `IntBits.h`, line 95 (see the patch).

Problem: "Segmentation fault" when running dictionary-related tests.
Cause: Results of string operations (like "`std::string::find_last_of`"), which are of type "`size_t`", were stored in variables of type "`unsigned int`" and then compared to the constant "`std::string::npos`", which is also of type "`size_t`". The type "`size_t`" is defined as "`unsigned long`" and the constant is equal to "`(unsigned long)-1`". Therefore, the result of the following comparison: "`n == std::string::npos`", where n is of type "`unsigned int`", will always be false. The problem doesn't exist on IA32, because on this platform both "`unsigned int`" and "`unsigned long`" are the same types (32-bit). On IA64 the former is 32-bit and the follower is 64-bit.
Remedy: in many files in the `src/Dictionary` directory all occurrences of "`unsigned`

10 All the patches (for SEAL, POOL, PI, Gaudi, Anaphe and MySQL++) are put in the following location: /afs/cern.ch/user/m/mkapalka/public/ia64-patches.tgz

int" have to be changed to "unsigned long" (see the patch). Actually, it would be probably better to change them to "size_t" instead. The patch also changes all "int" types to "long int". However, now it seems to be not always necessary and I think it can even cause some other problems (like in POOL).

Problem: "Segmentation fault" when running Minuit-related tests.
Cause: The reason is similar to the previous one – passing variables of type "int*" to functions converted from Fortran while their "integer*" type is actually "long int*". This makes a difference on IA64, while not being a problem on IA32. On IA64 it results in memory corruption, which is very difficult to debug.
Remedy: Changed a few Minuit files (see the patch).

Problem: in test_SealZip_MD5Digest01 the computed MD5 sum is different on IA32 than on IA64. What's strange is that both sums are incorrect (different than the output of the md5sum program)
Cause: type UINT4 which obviously should be 4-bytes long is defined as "unsigned long int".
Remedy: changed the type definition of UINT4 from "unsigned long int" to "unsigned int" in file src/Foundation/SealZip/src/ext/rfc1321/global.h. Now the sums computed on IA64 are the same as on IA32, but still they both differ from the output of the md5sum program.

3) Compilation Warnings

Problem: cast from pointer to integer of different size
Remedy: this shouldn't cause any problems as the warning appears when the following type of expression is used (note the second line):

```
_c_.addField("thePabs", "float", "",
                 (int)(&((PSimHit*)64)->thePabs)-64, PRIVATE );
```

The 3rd parameter of the "addField" function is a relative position of a given field within a class. This will do the right computations (I've checked that it really works).

Problem: a lot of warnings of type "unused parameter" or "defined but not used".
Remedy: shouldn't be dangerous, so I won't check it.

Problem: in SealBase/Signal, quite a few warnings like the following one:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/sr
c/Signal.cpp:1049: warning: int format, different type arg (arg 3)
```

Cause: the following instruction:

```
write(fd, buf, sprintf(...) );
```

The “write” function takes a variable of type “size_t” (unsigned long) as its third argument, while sprintf returns “int”.
Remedy: sprintf should always return positive numbers, so this shouldn't be a problem.

Problem: a few warnings like the following one:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/sr
c/Signal.cpp:1135: warning: unsigned int format, different type arg
(arg 4)
```

Cause: the following instruction:

```
  write (fd, buf, sprintf (buf, "  stack  = (%x, %x, %p)",
           uc->uc_stack.ss_flags,
           uc->uc_stack.ss_size,
           uc->uc_stack.ss_sp));
```

The field “uc_stack.ss_size” is “size_t” (unsigned long) and “%x” modifier corresponds to “unsigned int”.
Remedy: it also shouldn't be a problem, but I've made the small change: “%x” to “%lx”.

Problem: in SealBase/Socket a lots of warnings like the following:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/sr
c/Socket.cpp:169: warning: invalid conversion from `int*' to
`socklen_t*'
```

Cause: the following type of code:

```
SOCKOPT_LEN_TYPE  length = sizeof (data);
if (::getsockopt (SOCKETFD (), level, option, (char *) &data,
  &length) < 0)
throw NetworkError ("getsockopt()", ERRNO);
```

The type “SOCKOPT_LEN_TYPE” is set by configure script to "int", while the "length" parameter should be of type "socklen_t", which is 32-bit “unsigned int”.
Remedy: To suppress the warnings, I had to manually change the file src/Foundation/SealPlatform/src/network.m4 (see the patch) and then rebuild the configure script:

```
aclocal -I src
mv aclocal.m4 src
autoconf -Isrc
```

Problem: bit operations cause the following warnings:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h: In instantiation of
`seal::BitPatternHelp<2>::PatWrapper<65535, 32>':
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h:272:   instantiated from `seal::BitPattern<65535,
32, 2>'
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h:272:   instantiated from
`seal::BitOpsMagic<16>::Type<size_t>'
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h:272:   instantiated from
`seal::BitOpsCeil2<16>::Op<size_t>'
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h:272:   instantiated from `static T
seal::BitOpsCeil2<B>::Op<T>::compute(T) [with T = size_t, unsigned
int B = 32]'
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h:286:   instantiated from `static T
seal::BitOps<T>::ceil2(T) [with T = size_t]'
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealIOTools
/src/ReadBuffer.cpp:207:   instantiated from here
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Foundation/SealBase/Se
alBase/BitOps.h:272: warning: left shift count >= width of type
```

And the same for `ReadWriteBuffer.cpp`:336.
Remedy: using the Vladimir's solution:

```
- size_t      newsize = BitOps<size_t>::ceil2 (oldsize + n);
+ size_t      newsize = BitOps<int>::ceil2 (oldsize + n);
```

Problem: in PyLCGDict/MethodDispatcher:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Scripting/PyLCGDict/sr
c/MethodDispatcher.cpp:63: warning: int format, different type arg
(arg 3)
```

Cause: the following line:

```
sprintf( txt, "None of the %d overladed methods succeded",
m_methods.size());
```

Remedy: not dangerous.

Problem: in PyROOT:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Scripting/PyROOT/PyROO
T/CTypePtr.h:167: warning: no return statement in function
returning non-void
```

Remedy: this function always throws an exception, so the warning is irrelevant.

Problem: some warning in tests:

```
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/MathLibs/Minuit/tests/
MnSim/PaulTest.cpp:68: warning: comparison between signed and
unsigned integer expressions
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/MathLibs/Minuit/tests/
MnSim/PaulTest2.cpp:54: warning: comparison between signed and
unsigned integer expressions
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/MathLibs/Minuit/tests/
MnSim/PaulTest3.cpp:53: warning: comparison between signed and
unsigned integer expressions
/data1/mkapalka/packages/SEAL/SEAL_1_3_4/src/Scripting/PyLCGDict/te
sts/dict/myclass.h:107: warning: deprecated conversion from string
constant to `char*'
```

Remedy: all PaulTests fail anyway (also on IA32), because they need some input parameters and I have no idea what to pass to them. I didn't waste time on that.

4) Remaining Issues

Although many issues have been solved, some tests still fail. Surprisingly, some of them fail also on IA32[11], what means that they are not very well maintained and already outdated. Therefore, what we should worry about are only the tests that show some IA64-specific problems. These are the following:

- `test_PyROOT_basics.py` – complains about missing "PyROOT" module (this should be easy to solve as soon as I find the missing module),
- `test_SealBase_LocalServerSocket01` – failure,
- `test_SealBase_TimeInfo01` – complains about real time being smaller than CPU time (it's a bug),
- `test_SealKernel_Exception` – the output on IA32 is slightly different that on IA64. One has to look closer at that,
- `test_SealZip_CPIOOutputStream0{0|2|3}` – failure.

None of these issues look like very critical bugs, so I proceeded with other LHCb-related packages. However, one should repair these bugs sooner or later.

3.5 POOL

1) Compiling and Testing POOL

Unpack the source and next, after adjusting the configuration files[12], like it was with

11 See /afs/cern.ch/user/m/mkapalka/public/seal-test-results.tgz for the full output of all the SEAL tests on IA32 and IA64

12 See /afs/cern.ch/user/m/mkapalka/public/pool-ia64-config.tgz for an example configuration

SEAL, type:

```
cd POOL/POOL_1_6_3/src
source ../setenv
scram b
```

Unfortunately, with POOL one cannot just type "`scram b release-build`" and "`scram b release-check`" (like it was with SEAL), nor "`scram b release`" – these targets are not defined. Therefore, the build & test process is more complicated.

The testing procedure is more convenient when the oval tool is used. To run a test one has to go to its source directory (where `OvalFile` is put) and type "`oval prod`". This will firstly run the test and then compare its output to an appropriate reference file. If there are any differences, they will be shown. But before that, one has to set up the testing environment:

```
cd POOL/POOL_1_6_3/src
source ../pool_settstenv
```

It's also worth noting that many of the tests run only when all `*.root`, `*.pool`, etc. files are deleted before – they cannot overwrite the files and so they complain about not being able to open their output files for writing. Pretty annoying...

The Intel compiler has some nice portability warnings enabled by: "`-Wcheck`" and "`-Wp64`" options (see the manual). Also "`-Wall`" and "`-w2`" can make it produce much more warnings than would a GNU compiler for the same source files. This can be quite helpful in detecting all the IA64 issues, although it doesn't, unfortunately, discover all the problems (e.g. explicit casts from "`long`" to "`int`", even if they are semantically wrong, won't be signalized for obvious reasons). Unfortunately, POOL doesn't have a built-in configuration file for icc, so I've created one[13]. The problem is that not all the files will compile with the Intel compiler – some of them have some constructs which are causing compilation errors. Nevertheless, the aim was not to produce executables with icc, but to have some warnings – and this has been achieved.

The files should be downloaded and unpacked in the POOL directory. The `SCRAM_ARCH` environmental variable should be set to "`rh73_icc71_dbg`" (I actually use icc 8.0, but there were already some few configs for 7.1, so I just used them and extended a bit). Next the "icc" tool should be added to scram:

```
scram setup -i icc 7.1 file:./.SCRAM/ToolFiles/icc_7.1
scram tool list (check that "icc" is on the list)
```

Another problem with icc is that it can produce too many irrelevant warnings. Therefore, its output should be well filtered before it's actually analyzed by anyone.

13 Can be found at: /afs/cern.ch/user/m/mkapalka/public/pool-icc-config.tgz

2) Compilation and Run-Time Errors

Problem: quite a lot of errors of type:

```
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/StorageSvc/src/DbDatab
aseObj.cpp:399: no match for `seal::MessageStream& <<
std::ios_base&(&)(std::ios_base&)' operator
```

Cause: the problem is with constructs like “`log << std::hex`” where “`log`” is of type “`MessageStream`”.
Remedy: SEAL has to be patched (not POOL!). I've found a patch in the SEAL bug database (file `SealKernel/MessageStream.h`). It's already included in my patch.

Problem: configuration problems with MySQL++ headers – a few errors of type:

```
/data1/mkapalka/build/mysql++/include/defs:5:19: mysql.h: No such
file or directory
```

Cause: MySQL puts its header files in `$BUILDDIR/mysql/include/mysql` by default but POOL looks for them in `$BUILDDIR/mysql/include`. The same problem is with libraries.
Remedy: Copy, link or move the files to appropriate directories.

Problem: MySQL++ once again – in many files the following problem appears:

```
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/MySQLCatalog/src/MySQL
FileCatalog.cpp: In member function `virtual void
pool::MySQLFileCatalog::insertPFN(pool::PFNEntry&) const':
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/MySQLCatalog/src/MySQL
FileCatalog.cpp:464: ISO C++ says that `std::basic_ostream<char,
_Traits>& std::operator<<(std::basic_ostream<char, _Traits>&, const
char*) [with _Traits = std::char_traits<char>]' and `SQLQuery&
operator<<(SQLQuery&, const mysql_ColData<std::string>&)' are
ambiguous even though the worst conversion for the former is better
than the worst conversion for the latter
```

Cause: in the following code the “<<” operator is ambiguous:

```
Query q;
...
q << "a query string...";
```

Remedy: need to explicitly cast “`Query`” to “`SQLQuery`” before the “<<” operator is used, like the following:

```
(SQLQuery)q << "a query string...";
```

Problem: while linking EDG tests with the EDG libraries:

```
/data1/mkapalka/packages/POOL/POOL_1_6_3/rh73_gcc32_dbg/lib/liblcg_
EDGCatalog.so: undefined reference to `__ctype_b'
```

Cause: the symbol "`__ctype_b`" is defined in `libc-2.3.2.so` and is no longer defined in later versions of this library. This is a compatibility problem between different `glibc` versions and it appears when one uses "`__ctype_b`" directly (like in `EDGCatalog`). It's an internal symbol, so it should not be used by any applications, but here it is, so we have a problem.
Remedy: Andreas has recompiled the EDG libraries with the newest `glibc` library and this works perfectly now.

Problem: "`unsigned int`" vs. "`size_t`" – the same story once again:

```
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/DataSvc/src/Cache.h:75
: conflicting return type specified for `virtual unsigned int
pool::Cache::cacheSize()'
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/DataSvc/src/ICache.h:8
5:   overriding `virtual size_t pool::ICache::cacheSize()'
```

Cause: a virtual method "`cacheSize`" is defined as returning "`unsigned int`" in parent class and "`size_t`" – in a descendant class. This makes difference only on 64-bit architectures.
Remedy: changed "`unsigned int`" to "`size_t`" in files: `Cache.h` and `Cache.cpp`. However, CacheSVC and ICacheSVC both use "`unsigned int`" instead of "`size_t`", so they compile well, but it might cause some problems in future – we'll see...

Problem: lcgdict complains:

```
In file included from /
data1/mkapalka/packages/POOL/POOL_1_6_3/src/Tests/DataSvc_CrossRefe
rence/dict/allHeaders.h:1:
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/DataSvc/DataSvc/AnyPtr
.h: In
   constructor `pool::AnyPtr::AnyPtr(const T*) [with T =
RelatedClass]':
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/DataSvc/DataSvc/Ref.h:
189:   instantiated from `pool::Ref<T>& pool::Ref<T>::operator=(T*)
[with T = const RelatedClass]'
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/Tests/DataSvc_CrossRef
erence/src/CrossReferenceClass.h:34:   instantiated from here
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/DataSvc/DataSvc/AnyPtr
.h:21: error: invalid
   use of undefined type `struct RelatedClass'
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/Tests/DataSvc_CrossRef
erence/src/CrossReferenceClass.h:7: error: forward
   declaration of `struct RelatedClass'
Parsing file /
data1/mkapalka/packages/POOL/POOL_1_6_3/src/Tests/DataSvc_CrossRefe
```

```
rence/dict/allHeaders.h with GCC_XML Error processing file with
gccxml
```

Remedy: it's a known POOL bug (#3062), so an appropriate patch is available.

Problem: while creating `libDataSvc_RootSvcPerformanceDict.so`:

```
/data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Tes
ts/DataSvc_RootSvcPerformance/dict/../src//capabilities.o
(.text+0x0): In function `SEAL_CAPABILITIES':
/data1/mkapalka/build/seal/src/Dictionary/ReflectionBuilder/Reflect
ionBuilder/SealCapabilities.h:17: multiple definition of
`SEAL_CAPABILITIES'
/data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Tes
ts/DataSvc_RootSvcPerformance/dict/../src//capabilities.o
(.text+0x0):/
data1/mkapalka/build/seal/src/Dictionary/ReflectionBuilder/Reflecti
onBuilder/SealCapabilities.h:17: first defined here
```

Cause: it's because the `capabilities.o` file is given twice in the command line.
Remedy: the library can be linked manually:

```
/usr/bin/c++ -o /
data1/mkapalka/packages/POOL/POOL_1_6_3/rh73_gcc32_dbg/tests/lib/li
bDataSvc_RootSvcPerformanceDict.so -shared  /
data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Test
s/DataSvc_RootSvcPerformance/dict/../src/RandomDataDump_dict.o  /
data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Test
s/DataSvc_RootSvcPerformance/dict/../src/DataDump_dict.o  /
data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Test
s/DataSvc_RootSvcPerformance/dict/../src/RandomDataDump_dictstubs.o
/
data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Test
s/DataSvc_RootSvcPerformance/dict/../src/DataDump_dictstubs.o /
data1/mkapalka/packages/POOL/POOL_1_6_3/tmp/rh73_gcc32_dbg/src/Test
s/DataSvc_RootSvcPerformance/dict/../src//capabilities.o -L/
data1/mkapalka/packages/POOL/POOL_1_6_3/rh73_gcc32_dbg/lib -L/
data1/mkapalka/packages/POOL/POOL_1_6_3/rh73_gcc32_dbg/tests/lib
-L/data1/mkapalka/build/uuid/lib -L/data1/mkapalka/build/boost -L/
data1/mkapalka/build/root/lib/root -L/data1/mkapalka/build/pcre/lib
-L/data1/mkapalka/build/seal/rh73_gcc32_dbg/lib -lStressTestBase
-llcg_DataSvc -llcg_PersistencySvc -llcg_StorageSvc -llcg_POOLCore
-llcg_SealBase -llcg_PluginManager -llcg_SealKernel
-llcg_ReflectionBuilder -llcg_Reflection -llcg_FileCatalog
-llcg_AttributeList -llcg_Collection -llcg_CollectionBase -luuid
-lCint -lCore -lHist -lGpad -lGraf -lMatrix -lPhysics -lPostscript
-lTree -lpcre -lnsl -lcrypt -ldl -Wl,-E
```

Problem: most of the tests fail with the following error:

```
/data1/mkapalka/packages/POOL/POOL_1_6_3/src/StorageSvc/src/DbHeap.
cpp:228: static DbObject* pool::DbHeap::allocate(long unsigned int,
pool::DbContainer*, const pool::DbLink*,
pool::DbObjectHandle<DbObject>*): Assertion `sizeof(DbObjectGuard)
<GUARDSIZE' failed.
```

Cause: as the class "`DbObjectGuard`" contains some pointers and "`size_t`" variables, its size has increased while moving from IA32 to IA64. The constant "`GUARDSIZE`" is hard-coded to 64.
Remedy: I've changed `GUARDSIZE` in `StorageSvc/src/DbHeap.cpp` to 128. It might have been better to compute the value, based on the size of `DbObjectGuard` – to be considered in the future.

Problem: many test fail with "segmentation fault".
Cause: many reasons, mainly related to the difference between "`int`" and "`long`" (or "`unsigned int`" and "`unsigned long`") on IA64.
Remedy: I've prepared appropriate patches that solve most of the problems. However, this is an ad-hoc solution, as many unanswered questions appeared – but it's up to Markus Frank and other POOL developers to think about them.

3) Remaining Issues

Well, most of the POOL tests work well. It's difficult to say, how large is their code coverage, but hopefully they test most of the needed functionality. Nevertheless, some things are still causing problems and there is probably no better way to repair them than detailed analysis of the whole source tree and debugging the crashing programs. It may also require rethinking some of the assumptions, e.g. answering the question: where are "longs" really needed and where "ints" are just sufficient. The following test programs fail at the moment:

- `PersistencySvc_Functionality` – it works, but the output is slightly different than in the reference file: "database 2 size : 20 kB" instead of "database 2 size : 19 kB",
- `DataSvc_CMSMultiCache` and `DataSvc_PtrOwnership` tests hang,
- `DataSvc_ResetObject` – finishes with some errors,
- `DataSvc_RootSvcPerformance` – fails (an exception thrown),
- `MultiCollection_BasicFunctionality` – "segmentation fault",
- `ExplicitCollection_Functionality` – hangs,
- `Collection_*` tests: `Read`, `Write`, `Update`, `MultiFileWrite`, `MultiFileUpdate`, `Explicit{Read|Write}Performance` are ok, but their output is slightly different than in the reference files; however, in my opinion the differences should not cause any problems – but this, of course, should be judged by a more competent person,
- `Collection_FileInfoRetrieve` – "segmentation fault",

- `FileCatalog_Functionality` – fails because it expects some data present in a MySQL database and I have no idea, what should be put there,
- `DataRegression_DataSvc_*` tests: `CMSCollDataModel` and `SimpleEmbeddedRefs` hang,
- `XMLCatalog/tests/XMLFunctionality` – runs well, but some differences from the reference file,
- `EDGCatalog/tests/EDGlookupTest` – ok, but difference from the reference file: the output is "bestpfn: replicapfn filetype: ROOT_All" instead of "bestpfn: pfntest filetype: text",
- `EDGCatalog/tests/EDGFunctionality` – fails with a message "wrong file type" (this might be related to the previous error),
- `RootCollection/tests/{read|update}` – "segmentation fault",
- `MySQLCatalog/tests/*` tests: `importTest`, `MySQLFunctionality`, `insertTest`, `lookupTest`, `metadataTest` don't fail, but one can find the following message within their output: "pool::FC::MySQL++Query in MySQLFileCatalog::insertPFN Query was empty Status=5", which is quite suspicious,
- MySQLCollection and MySQLltCollection related tests – they all fail. Probably a local MySQL database has to be configured somehow and has to have some tables already present. Unfortunately, I have no idea, how to prepare the run-time environment for these tests.

3.6 PI

1) Compiling and Testing PI

Firstly, adjust PI configuration files[14] (in the same way as for SEAL and POOL). Next, you can compile PI with the following commands:

```
cd PI/PI_1_1_3/src
eval `scram runtime -sh`
scram b release-build
```

The "`scram runtime -sh`" seems to be easier to use than my "`setenv`" and "`settstenv`" scripts, although it works equally well. To test PI, one can use the "`scram b release-check`" command. However, it produces quite a huge file, which is difficult to analyze. My method was to run all the test programs from the "`rh73_gcc32_dbg/bin`" directory one by one and see, what's happening (e.g. compare the outputs on IA32 and IA64).

14 See /afs/cern.ch/user/m/mkapalka/public/pi-ia64-config.tgz

2) Compilation Errors

Problem: `libpacklib.a` is needed, but the path `/cern/pro/lib/libpacklib.a` is hard-coded in a makefile.
Remedy: edit `src/AnalysisServices/AIDA_HBookStore/src/BuildFile` and change:

```
- CERNLIB = -static ...
+ CERNLIB = -static $BUILDDIR/cernlib/lib/libpacklib.a
$BUILDDIR/cernlib/lib/libkernlib.a -L$BUILDDIR/libshift/lib -lnsl
-lcrypt -ldl -lshift
```

Of course, it's better to replace `$BUILDDIR` with an appropriate patch manually, as I'm not sure if it will be substituted at compilation time.

Problem: the compilation error:

```
/data1/mkapalka/packages/PI/PI_1_1_3/rh73_gcc32_dbg/lib/liblcg_AIDA
_HBookStore.so: load failed:
Shared library operation dlopen() failed because: libshift.so:
cannot open shared object file: No such file or directory
```

Remedy: add `$BUILDDIR/libshift/lib` to `LD_LIBRARY_PATH`.

Problems: an undefined symbol:

```
/data1/mkapalka/packages/PI/PI_1_1_3/rh73_gcc32_dbg/lib/liblcg_Plug
inFaFNative.so: undefined symbols
  intrac_
```

Remedy: if you followed all the steps with Anaphe, it shouldn't occur. It was solved by using static `libpacklib.a` instead of `libminuit.so` in Anaphe.

Problem: some files missing (a configuration problem, as the files are present in the directory tree):

```
/data1/mkapalka/packages/PI/PI_1_1_3/src/AnalysisServices/DataXML/t
ests/dxml_test.cpp:1:25: DataObject.h: No such file or directory
/data1/mkapalka/packages/PI/PI_1_1_3/src/AnalysisServices/DataXML/t
ests/dxml_test.cpp:2:24: XMLStream.h: No such file or directory
```

Remedy: probably one should edit the configuration file `src/AnalysisServices/DataXML/tests/GNUmakefile`. However, I haven't managed to make it work – actually I haven't spent too much time on that as this is only a test program, so it's not that important.

Problem: a very common error on IA64:

```
/usr/bin/ld: /data1/mkapalka/build/cernlib/lib/libpacklib.a
(cfopei.o): @gprel relocation against dynamic symbol cfopen_perm
```

Cause: it means that the library (`libpacklib.a`) hasn't been compiled/linked with “`-fpic`” option.
Remedy: the CERNLIB has to be compiled with “`-fpic`” option – if you followed the steps in previous sections, it's already done.

3) Remaining Issues

A few tests don't want to compile. But the question is if we really need them. If yes, it might be needed to tune up the configuration files, as they are probably causing all the problems. On the other hand, most of the tests that compile run well. All the tests that fail on IA64, fail also on IA32, so there is no problem. However, some output numbers and output files are slightly different on IA64 than on IA32[15] – maybe it will be necessary to trace this problem further in the future.

3.7 Gaudi

1) Compiling and Testing Gaudi

Firstly, all the relevant packages have to be copied from `$BUILDDIR` and `$PACKDIR` (SEAL, POOL and PI) to `/data1/lhcb/sw/packages`. The structure of this directory should be the same as on `/afs/cern.ch/sw/packages`[16] (it may require some file movements). Next, one should set up the LHCb and Gaudi environment and start building the libraries (this is valid on `oplapro49`):

```
tcsh
source /afs/cern.ch/lhcb/scripts/lhcbenv.csh
cmt64

setenv GAUDIDIR /data1/lhcb/GAUDI/GAUDI_v14r5 (for our convenience)
GaudiEnv v14r5
cd $GAUDIDIR/Gaudi/v14r5/cmt
cmt show uses
source setup.csh
cmt broadcast gmake
```

The Gaudi examples can be compiled with the following command (after setting up the environment as for compiling Gaudi itself):

```
cd $GAUDIDIR/GaudiExamples/v14r1/cmt
source setup.csh
```

15 See /afs/cern.ch/user/m/mkapalka/public/pi-test-results.tgz for the output of all the PI test programs on both IA32 and IA64

16 See /afs/cern.ch/user/m/mkapalka/public/dirstruct.tgz

```
gmake
```

To run one of the examples:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:
   ${BUILDDIR}/libshift/lib:
   ${GAUDIDIR}/GaudiSvc/v11r6p1/cel3-ia64_gcc323:
   ${GAUDIDIR}/GaudiExamples/v14r1/cel3-ia64_gcc323
../cel3-ia64_gcc323/AlgSequencer.exe ../options/AlgSequencer.opts
```

Other examples can be run run in the same way. Their input parameters are in the `options` directory and the reference output files are in the `home` directory (both of them under `$GAUDIDIR/GaudiExamples/v14r1`).

To debug the examples that fail one might need to compile Gaudi with debugging information. This can be done in the following way (note, that the creating symbolic links part is needed only once – and it's needed because of a silly bug in one of the configuration files):

```
setenv CMTCONFIG $CMTDEB
bash (I just got used to this shell)
for i in `find /data1/lhcb/sw/packages -name cel3-ia64_gcc323`;\
   do echo $i; ln -s $i `dirname $i`/cel3_ia64_gcc323; done
exit
cmt broadcast "make clean"
cmt broadcast gmake
```

2) Compilation and Run-Time Errors

Problem: linker: cannot find `libshift`
Remedy: so obvious, but has to be done many times, as Gaudi configuration scripts overwrite the `LD_LIBRARY_PATH` variable:

```
setenv LIBRARY_PATH $BUILDDIR/libshift/lib
```

Problem: linker: cannot find `libRFIO`
Remedy: ROOT has to be built with RFIO support added (see previous sections).

Problem: problems with Python:

```
File "/data1/lhcb/sw/packages/SEAL/SEAL_1_3_4/cel3-
ia64_gcc323/bin/lcgdict.py", line 1, in ?
import sys, os, gendict, selclass, string, getopt
ImportError: No module named os
```

Remedy: we have to advice Python to use the right directory:

```
setenv PATH $BUILDDIR/python/bin:$PATH
setenv PYTHONHOME $BUILDDIR/python/
```

Problem: linker: cannot find `libpython2.2`
Remedy: simply:

```
cd /data1/lhcb/sw/packages/Python/2.2.2/cel3-ia64_gcc323
ln -s lib/libpython2.2.so libpython2.2.so
```

Problem: in `GaudiPoolDb` and other modules:

```
../src/PoolDbAddress.cpp:103: cannot convert `const unsigned int*'
to `const long unsigned int*' in return
```

Cause: POOL has changed (i.e. was ported), so some return types have also changed and obviously it will cause problems in every package that uses POOL directly.
Remedy: well, I had to convert a lot of things, as it was with POOL... Hard work, but after that Gaudi has successfully compiled. Of course, I've prepared a patch, but it's an ad-hoc solution for this particular version of Gaudi and POOL and should be rather considered as a didactic example and a base for further, complete and overall corrections of the newest versions of the packages.

3) Summary of the Test Results

There are only few examples shipped with Gaudi and it's difficult to say to which extent they test the whole framework. Nevertheless, they are pretty easy to use and to check, as they are option and reference files available. These are the results:

- `AlgSequencer` – OK,
- `AlgTools` – OK,
- `ColorMsg` – OK,
- `GaudiMT` – I couldn't run it and there is no reference file (probably the example is somehow outdated or requires some special treatment),
- `Gpython` – "segmentation fault",
- `GSLTools` – seems to be OK, but there is no reference file,
- Histograms – fails,
- `Ntuples` – fails,
- Properties – OK,
- `RandomNumber` – "segmentation fault",
- `RootIOWrite` – some errors,
- `RootIORead` – output different than the reference file (but the input file is produced by `RootIOWrite`, which fails, so that might be the reason).