View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CERN Document Server

EUROPEAN LABORATORY FOR PARTICLE PHYSICS

CN / 95 / 13

11 September 1995

Performance Awareness

(Execution performance of HEP codes on RISC platforms, issues and solutions)

Refael Yaari

Weizmann Institute
Israel
(FHYaari2@Weizmann.Weizmann.ac.IL)

Sverre Jarp

Computing and Networks Division / CERN
1211 Geneva 23 Switzerland
(Sverre.Jarp @ Cern.CH)

TABLE OF CONTENTS:

ABSTRACT **3**
ACKNOWLEDGEMENT **3**
1 INTRODUCTION **4**
1.1 IN PERSPECTIVE 4
1.2 WHY IS CODE OPTIMISATION STILL OF INTEREST ? 4
1.3 PERFORMANCE AWARENESS 5
1.4 COLLABORATION WITH ALEPH 5
1.5 COMPILER OPTIONS ACROSS PLATFORMS 6
2 PERFORMANCE OF THE MAIN ALEPH PROGRAMS **7**
2.1 THE FIRST PROFILING EXERCISE: GALEPH 7
2.2 STUDY OF THE RNDM FUNCTION AS USED IN GALEPH 8
2.3 STUDY OF THE IUHUNT FUNCTION CALLS IN GALEPH 9
2.4 GALEPH RUNS WITH MODIFICATIONS 10
2.5 ANALYSIS OF JULIA 11
2.6 ANALYSIS OF ALPHA 12
3 KERNEL STUDIES **14**
3.1 INTRODUCTION 14
3.2 BIT SHIFTING 14
3.3 BIT TESTING 14
3.4 AN ANALYSIS OF DSIGN 15
3.5 MEASURING THE WHOLE SET OF ARITHMETIC FUNCTIONS 16
4 FREQUENTLY USED CERNLIB ROUTINES **17**
5 REALISTIC AND OPTIMISTIC BENCHMARKING **17**
6 GEXAM1 MEASUREMENTS **19**
7 PROFILING OF ATLAS CODE **20**
7.1 INTRODUCTION 20
7.2 THE ATLAS SIMULATION PROGRAM (DICE) 20
7.3 RUNNING ATRECON 21
8 CONCLUSIONS AND RECOMMENDATIONS **22**
REFERENCES **23**
APPENDICES **24**
I - MACHINE SPECIFICATIONS **24**
II - GRNDM STUDY **24**
III - SOURCE CODE OF THE RANDOM NUMBER GENERATORS **25**
IV - SOURCE CODE OF IUHUNT **26**
V - KERNEL TABLES **29**
VI - BENCHMARK TABLES **33**

VII - GEXAM1 MEASUREMENTS **33**
VIII - VZERO AND UNROLLING **35**
IX- PROFILING FULL DICE CODE **37**

Abstract

The work described in this paper was started during the migration of Aleph's production jobs from the IBM mainframe/CRAY supercomputer to several RISC/Unix workstation platforms. The aim was to understand why Aleph did not obtain the performance on the RISC platforms that was "promised" after a CERN Unit comparison between these RISC platforms and the IBM mainframe. Remedies were also sought.

Since the work with the Aleph jobs in turn led to the related task of understanding compilers and their options, the conditions under which the CERN benchmarks (and other benchmarks) were run, kernel routines and frequently used CERNLIB routines, the whole undertaking expanded to try to look at all the factors that influence the performance of High Energy Physics (HEP) jobs in general.

Finally, key performance issues were reviewed against the programs of one of the LHC collaborations (Atlas) with the hope that the conclusions would be of long-term interest during the establishment of their simulation, reconstruction and analysis codes.

Acknowledgement

First of all we are indebted to R.Hagelberg, E.Lançon, J.Knobloch and F.Ranjard from Aleph for their help and encouragement. Likewise, we would like to thank A.Dell'Acqua, M.Nessi and G.Poulard from Atlas. We received a lot of help from the various manufacturers on site and we should thank J-P.Thibonnier and C.W.Hobbs from DEC, F.Rademakers from HP, T.Bell from IBM, and I.Zacharov from SGI. F.Hemmer and L.Robertson (PDP/CN) provided support for R.Yaari's one-year stay at CERN and C.Boissat (PDP/CN) was extremely helpful in making sure our benchmark jobs ran easily in the SHIFT/CORE environment.

Finally several people, both in CN and elsewhere, help get the final report put together.

1. INTRODUCTION

1.1 In perspective

Recently we have been observing an explosion of affordable processing-power across the computer industry. CMOS[1]-based RISC microprocessors with processing power that exceed that of expensive mainframes are now available at a cost well below $10,000 for a complete workstation. In High Energy Physics (HEP) the consequences have been extremely positive, especially given the fact that no previous efforts (such as vectorisation) had really managed to revolutionise the cost of HEP computing.

The question raised in this paper is the issue of code optimisation. With a 15-20 million dollar CDC 7600 (rated at 10 MIPS) or IBM 370/168 (whose capacity incidentally helped define the CERN unit at 4 MIPS), there was no doubt that code optimisation was justified based on the fact that the acquisition costs (even before adding maintenance) corresponded to hundreds of man-years. As a consequence, even if it took a complete man-year for somebody to improve the annual output of such a machine by a modest 1% - the effort was already cost-justified.

Today, the situation is quite different. A 20 processor SGI Challenge (of the type used by L3) with an estimated performance of 600 CERN units and hundreds of Gigabytes of disk space can be acquired for about 1 million dollars. Given an annual manpower cost above $100,000 it is not difficult to understand that the situation has changed dramatically. A programmer would now be required to make improvements per man-month that has a global effect of several percent.

1.2 Why is code optimisation still of interest ?

The LEP experiments, which were promised no more than 2 CERN units each by the CERN Directorate in 1982 but now enjoy more than 100 times that tiny capacity, are still not satiated. Even before the LHC experiments started using CERN's Central Simulation Farm (CSF), one or two LEP experiments still managed to swamp the whole farm without any problems. This was seen as a clear indication that the need for simulation capacity is unlimited[2]. As the accumulated number of Z^0 events increases until the end of LEP-100, it is also expected that the capacity for analysis will continue to grow.

Another argument for code optimisation comes from the fact that physicists will always welcome better turn-around. If a job that used to run in a day, can now be made to take only one hour, so much the better. Unfortunately, whereas industry can often relate to direct or indirect manpower savings in such cases (to justify the tuning effort), a research organisation like CERN tends to categorise such improvements as increased "comfort" rather than improved efficiency.

New experiments, such as the heavy ion NA49 experiment and the LHC experiments in the next decade, will have much bigger appetites than LEP. A fully simulated NA49 event is estimated to take about 1 hour on a CSF processor and a full simulation of a LHC event may last 100 times longer than a LEP event.

By gradually changing the way physicists perform data analysis, one can also expect large additional CPU capacities to be required. When hundreds of LEP and LHC participants start performing data mining on huge event loops in quasi-real-time on super-PIAF systems, these will have to be of enormous capacity and extremely well tuned.

Other paradigm shifts are also coming along, be it virtual walks inside an operational detector

---

[1] Composite Metal-Oxide Semiconductor

[2] Today several LEP experiments have private simulation farms in addition to using the central cluster.

or near real-time simulation of the LHC accelerator that will require enormous amounts of well-optimised computing capacity.

Another factor of influence will be the rate at which manufacturers offer higher CPU-speeds. Whereas it has been relatively straightforward to increase microprocessor chip speeds from say 20 to 40, 80 and 160 MHz, it is very likely that the continued doubling (to 320, 640 and higher) is going to be much more difficult, and the microprocessors may not show up as quickly as the HEP community has anticipated.

When the manufacturers use other "tricks" to increase their performance ratings, such as the inclusion of more and more parallel execution units, speculative execution, branch prediction, a more complex instruction set, multiple cache levels, etc., it is unavoidable that only with a corresponding tuning effort will we come close to optimal use of these complex additions to the original RISC concept. Of course, sophisticated compilers will try to close the gap, but it is highly unlikely that these compilers will grow in complexity sufficiently quickly.

1.3 Performance awareness

What is needed, is a "performance awareness" which takes into account both the manpower spent on improving performance and the hardware resources in use, both on a local and a global HEP scale. Therefore, a global change, such as the introduction of virtually divided volumes inside the GEANT source, has much more impact than, for instance, a change in an on-line reconstruction program, which is only running on one dedicated workstation in the pit.

If we try to define a hierarchy for performance enhancements, global algorithmic improvements that span experiments and/or machine architectures should always come first. Then come experiment specific changes or vendor-specific improvements (such as a new compiler, new math library). Lastly comes improvements that only have a short-lived effect. Typical examples would be the use of a compiler option or making some code changes to remove a limitation in cache-size, overcome the slowness of floating-point divide or square-root calculations on one particular RISC system. Even this last category has its merits when a collaboration has acquired large amounts of hardware resources and know that their programs will run on this hardware for several years.

We return to this classification in the conclusions in Chapter 8.

1.4 Collaboration with Aleph

The first part of the work covered in this report was triggered by the fact that during its transition from CERN's mainframes (the IBM ES/9000 and the Cray X/MP) the Aleph collaboration had noticed that when their main programs ran on RISC platforms the performance was not as good as expected in many cases. Their expectations were based on a straightforward CERN Unit (CU) comparison. At a meeting in January 1994, after some preliminary results confirming this fact were reproduced by CN[3], it was decided to launch a study to try to understand the reasons for it. The degradation was considered serious as it was well above a factor 2 in certain cases..

The RISC platforms in question were the DEC/3400 SAGA farm, the HP/9000/735 (99 MHz) Central Simulation Farm (CSF) and the SGI Challenge (150 MHz) in SHIFT. Later in the study new machines from each manufacturer were introduced, namely the DEC/3900, the upgraded HP/735 (125 MHz), and the upgraded SGI/Challenge (200 MHz). The programs used were GALEPH (the Monte Carlo simulation program, based on GEANT 3.15 which tracks e+e- events through the Aleph detector and digitises them for reconstruction), JULIA (the reconstruction program which tries to "build" the physics event from the raw data or from simulated data) and ALPHA/BT (the analysis program for B-tagged events which is a typical

[3] The Computing and Network Division in CERN.

LEP analysis). As opposed to the early investigations, which had been using only 10 events, we decided to run with a relatively large sample of events (461) in order to minimise statistical fluctuations.

Some months later (July 1994), starting from updated versions of the programs, new libraries, and presumably new compilers and system releases, we measured all three programs on the same platforms as before and also on an IBM/RS6K-350 in use by the MPI group in Aleph[4]. In the case of IBM we later added a top-of-the-range RS6K-590. The timing measurements of the production machines can be seen in Tables 3, 8 and 12 from which the following summary table was built:

Table 1: Execution Performance Summary (28/7/94)
(Timing ratios, compared to CERNVM[5])

	Platform					
	SAGA DEC/3400		CSF02 HP/735		SHIFT9 SGI/150 MHz	
	[18 CERN Units]		[27 CERN Units]		[20 CERN Units]	
Program	Original	Re-normalised	Original	Re-normalised	Original	Number of events
GALEPH	63 %	70 %	65 %	49 %	57 %	461
JULIA	52 %	58 %	112 %	83 %	93 %	461
ALPHA	39 %	39 %[6]	95 %	70 %	49 %	2000

Some of the programs were performing better on some of the platforms, in particular ALPHA on HP and JULIA on HP and SGI, but the Monte-Carlo runs still showed inadequate performance on HP and SGI. This was particularly disturbing since HP was meant to be the simulation farm "par excellence".

As a consequence, we decided to try to understand the reasons for some of these discrepancies and to point out some solutions that might improve the performance of the Aleph codes on RISC platforms. We also wanted to show how this gap could, eventually, be closed and suggest methods to improve performance of HEP programs in general. (See recommendation & conclusions in Chapter 8).

1.5 Compiler options across platforms.

Since we ran our tests on multiple platforms we needed to choose, at the very beginning, the same (or at least similar) levels of optimisation. Otherwise there would be no way to compare the timing measurements in a relatively fair way. This problem was exacerbated by the fact that even the default values from the vendors were completely inconsistent. For example, the O option on DEC's SAGA implied O4 (which was almost the highest level) whereas on SGI it

[4]Apologies to Sun which was left out of this study, mainly because there was no Aleph port available

[5]Since CERNVM (IBM/ES9000), was equivalent to 20 CERN units, each RISC platform had its timing re-normalised with the ratio between the CERNVM CERN units and its own CERN units. Note that the SGI, which is also considered 20 CU, is left untouched.

[6]According to Eric Lançon (Ref 1) this low number was due to the fact that most of the ALPHA library code was compiled with a low compilation level (-O1) which slowed down the performance by a factor of 1.7. It could have been 39% * 1.7 = 66% (a number more consistent with the other SAGA values). See section 2.8 for details.

meant O1, and on HP O0 (the absolute lowest level). It was clear that we could not use the defaults as assigned by the vendors. We had to choose an acceptable level, which would do more or less the same optimisation effort on each platform. The chosen value was O2. This choice was also related to the fact that experiments are often using this level to prevent unpleasant surprises from highly optimised code, which may give rise to different results on different platforms, not just due to different rounding schemes for floating-point numbers, but also due to possible errors which are still lurking inside the compiler. Table 2 summarises the situation:

Table 2: F77 optimisation levels on the various platforms

Platforms	Defaults	O	O0	O2	O3	O4
DEC	O = O4	–	Noopt	–	not for non-shared	
HP-CSF	O0	O1+O2	Const folding	O2	O3	
SGI	O1	O2	Noopt	ucode	Maxi	
RS-6000	NOOPT	O2	Noopt	O2	Maxi	
VM	OPT(3)	–	–	–	–	

2. PERFORMANCE OF THE MAIN ALEPH PROGRAMS

2.1 The first profiling exercise: GALEPH

First we established the full table of RISC to VM comparisons for the GALEPH Monte Carlo production job. Note that the IBM RS/6000-350 from the MPI collaboration is now also included.

Table 3: Total time for executing GALEPH on all platforms
(Number of events = 461)

Platform	Total time (seconds)	Relative to VM	%	Normalised to 20 CU (%)	I/O	System utilisation
VM	6622	1.00	100	100	full[7]	--
CSF02	9977	1.51	66	49	no	99%
SAGA01	10464	1.58	63	70	no	91%
Shift9	11582	1.75	57	57	no	99%
RS6K-350	15892	2.40	42	84	no	99%

The next task was to find out where the programs were spending the CPU cycles. In order to do so, we used the normal profiling utilities. Most Unix systems come with a standard set of performance tools. There are *prof* and *gprof* for execution profiling and calling-tree reporting.

[7] Due to disk space shortage the programs did not produce the binary EPIO output file, except on VM.

DEC and SGI offer *pixie* for statement basic-block tracing and instruction counts. IBM offers *tprof* whereas HP offers only a special performance packages (HP/PAK) as a separate product.

All vendors allow assembly-level code to be generated by the compilers for inspection, normally via a *-S* option, but such code can not necessarily be "trusted" in the case of SGI because there is a supplementary stage of optimisation of the machine code that takes place after the assembly code has been generated. In this cases, we are obliged to use the dissembler, *dis*, if there is a need to understand program behaviour at the lowest level.

When we looked at the execution profile of GALEPH we saw a consistent time distribution as demonstrated in Table 4.

Table 4: Time distribution in % for some of the GALEPH routines
(Number of events = 461; the symbol "-" means that the value is below threshold)

	SAGA	CSF	SHIFT9
rndm[8]	16.6	31.0	15.6
tpaval	4.0	3.0	4.9
wbank	5.4	4.0	-
tssham	-	3.2	2.0
gtnext	4.2	3.1	5.5
grndm	1.5	2.0	1.6
iuhunt	20.3	14.8	20.0

It was quite obvious that one routine, RNDM [See Ref. 4] was being heavily used and consumed most of the CPU cycles, in particular on the HP/735. A similar consumption was noticed in IUHUNT [See Ref. 4] on SAGA and SHIFT9 but to a lesser degree on HP. We decided to study the behaviour of these routines in stand alone kernel programs to determine the reasons for the "peaks" in the above distribution.

In addition to the CPU consumption we found that RNDM was called 2,800,000 times per event. IUHUNT, which was searching (inefficiently) for an element in a long array was called 28,000 times per event.

We decided that both routines were well worth a detailed study.

2.2 Study of the RNDM function as used in GALEPH

When we isolated the RNDM function in a stand alone program that just looped over two millions calls, we saw that on some of the RISC platforms, in particular on the HP/735, a pronounced performance degradation manifested itself. A slowdown of nearly a factor of 2.5, compared to CERNVM, can be seen from Table 5 where Rndm-i is the original RANECU [see Ref. 4] integer algorithm, Rndm-ftp is the REAL*8 version of the same algorithm, and Ranlux is the RANLUX (level 0) [see Ref. 4] algorithm adapted for RNDM usage.

[8] RNDM and GRNDM in Galeph are used in the following way: Inside GEANT code, the original GRNDM is used, preparing 100 random variables in an array, fetching 2-3 for each call. RNDM is called inside the GALEPH code, sharing the same array as in GRNDM, but fetching **only one variable** for every call from this array. This variant of RNDM, will correspond to what it referred to as "RNDM" from here onwards.

Table 5: Summary of Test-RNDM runs:
(Number of calls is always 2 million. All optimisation levels are O2)

Platform	Timing (in seconds) Rndm-i	Rndm-ftp	Ranlux	Ratios	Compiler
VM	3.12	3.82	1.80	1.7:2.1:1	
CSF02	7.24	3.84	1.26	5.7:3.1:1	
SAGA01	6.78	4.90	2.30	2.9:2.1:1	
RS-590	2.13	5.77	1.54	1.4:3.7:1	
SHIFT9	4.60	4.70	1.09	4.2:4.3:1	
SAGA01	2.93	3.98	1.07	2.7:3.7:1	New version 10/94[9]

HP-735 (CSF02) and the DEC/3400 (SAGA01) had the worst performance for the integer variant. On the HP, this could be attributed to the fact that integer multiply and divide are done by jumping to so-called millicode routines for execution, since there are no instructions for these operations[10]. By examining the code of the original RNDM routine, (Cf. Appendix III), we confirmed that the code is based on a pure integer algorithm for calculating new seeds and uses only one floating point operation to normalise the result to a value between 0 to 1. Altogether two integer divisions and six integer multiplications are used.

The floating-point variant shows an improvement of 1.4-1.8 for these platforms, due to the direct usage of floating-point instructions. This fact forced us to look in the CERN library for another suitable random generator that was floating-point oriented. We discovered RANLUX which has five luxury levels [See Ref. 4 for a description of all levels], and could easily be adapted to the GALEPH usage convention. (Cf. the RANLUX code in Appendix III).

The resulting performance of this function can be seen in the fourth column of Table 5. It shows a major improvement on all platforms, in particular on HP, DEC and SGI where factors of 5.7, 2.9, and 4.6 were achieved, respectively. Even on VM we observed an improvement of 1.7. From this we concluded that the algorithm used in RANLUX (with level =0), is highly efficient and might be considered for more heavy usage, since it apparently has all the needed qualities for a good random number generator[11].

Another approach to the RNDM was offered by somebody inside HP, US [Ref. 3], who found out that if all the constants parameters were stored in a common block, the compiler chose to invoke the XMPY instruction to perform integer multiplication. The results were less impressive than the RANLUX version, and since it was an example of only a "private" case for HP compilers, we felt reluctant to use it for further runs of GALEPH .

2.3 Study of the IUHUNT function calls in GALEPH

Investigating the way IUHUNT was used in GALEPH, we found calls in only three different places, from which 26K calls/event were made with an average array length of 1500 elements. After examination of the code (Cf. Appendix IV) we thought that there must be a better way of doing it. Our approach was to unroll the loop [See Ref. 6 and 8] with the hope that a sizeable improvement could be gained in execution time. As with RNDM we created a short test program

[9]The spectacular improvement seen with the new DEC compiler will be covered later.

[10]The HP does allow integer multiply, using the XMPY instruction, if the operands have been moved to floating-point registers (via the cache).

[11]Admittedly, we should have compared the generated physics, for instance the deposited energy distribution to make sure that RANLUX was good enough for Aleph.

that mimicked the calls from GALEPH and removed all other functionality. The size if the array was chosen to be 1500 corresponding to the peak of the distribution. Since it was possible to "hunt" through an array using different increments, we made runs with all increments used by Aleph (1,2, and 3). The results were as follows:

Table 6 : Summary of Test-IUHUNT runs:
(Number of calls is always 300. All optimisation levels are O2)

Variant 1- original version:

Increment	CERN VM	HP-735 99	HP--735 125	SHIFT9 SGI/150	SAGA01 DEC	RS-590 IBM
1	28.7 s	30.9 s	24.5 s	36.4 s	16.0 s	18.1 s
2	21.8 s	23.3 s	18.4 s	27.3 s	12.1 s	13.7 s
3	16.7 s	17.4 s	13.7 s	20.3 s	9.1 s	10.2 s

Variant 2- Unrolled version (Improvement ratio is given in brackets):

Increment	CERN VM	HP-735 99 MHz	HP--735 125 MHz	SHIFT9 SGI/150	SAGA01 DEC	RS-590 IBM
1	10.7 s (2.7)	15.3 s (2.0)	12.2 s (2.0)	16.7 s (2.2)	15.6 s (1.0)	17.3 s (1.1)
2	9.6 s (2.3)	11.7 s (2.0)	9.3 s (2.0)	12.7 s (2.2)	11.9 (1.0)	13.1 s (1.1)
3	7.2 s (2.3)	8.9 s (2.0)	7.2 s (1.9)	9.6 s (2.1)	9.1 (1.0)	9.2 s (1.1)

As can be seen, there was a pronounced improvement on almost all platforms giving a factor two on the first four, albeit almost unnoticeable on SAGA. This non-improvement indicates that the DEC compiler has done the unrolling by itself so we had no need to intervene. Quite unexpectedly, VM gained by the highest factor of all.

The listings of the original code and the unrolled version can be inspected in Appendix IV.

2.4 GALEPH runs with modifications

Once we realised that the RNDM and IUHUNT functions were the major source of the slowdown of GALEPH on the RISC platforms, we decided to rerun the measurements using the same input data, but with the following modifications:

- Use of the RANLUX code within the framework of RNDM and GRNDM.
- Replacing the standard IUHUNT with the unrolled version.

The results of these runs are shown in Table 7. Run_1 indicates the original measurement, before any modifications, Run_2 gives the timing when the RANLUX version of RNDM had been introduced and Run_3 adds in the unrolled version of IUHUNT as well. Ratio(i)/(j) is the ratio of run_i/run_j. "Perf_ratio" is the ratio of VM timing to RISC platform timing. This ratio is then normalised in "Norm_ratio" by dividing the performance ratio by the CERN Unit ratio taken from Appendix I.

Table 7: Total time for executing the modified GALEPH on all platforms
(Number of events = 461)

461 Events	CERNVM ES/9000	DEC/ 3400	HP-735 99 MHz	IBM RS-350	SGI 150 MHz
Run_1	6622 s	10263 s	9977 s	15852 s	12057 s
Run_2	5832 s	9770 s	7024 s	13782 s	10751
Ratio(1)/(2)	1.14	1.12	1.42	1.15	1.12
Run_3	5659 s	8873 s	5321 s	13681 s	8833 s
Ratio(2)/(3)	1.03	1.10	1.32	1.01	1.21
Perf_.ratio	1.0	0.64	1.06	0.41	0.64
Norm_.ratio	1.0	0.71	0.79	0.82	0.64

The HP-735 systems in CSF which had been a major worry was now at about 80% of "promised" performance even after applying the same improvement to CERNVM, whereas before the improvements, it was only 49% (Cf. Table 1). The improvements were less striking on the other platforms since RANLUX gave an improvement similar to what CERNVM obtained; i.e. no net change. It is not clear why SAGA showed an improvement when using the unrolled IUHUNT since the stand-alone run had determined that the compiler unrolled it by itself. It is assumed that the IUHUNT version coming from the library, was not unrolled after all.

Although this was an encouragement, we had still not reached the goal of fully explaining why Aleph was getting less relative performance on the RISC platforms. We will therefore come back later to where the rest of the "missing" percentages might be found.

2.5 Analysis of JULIA

The next task was to run the reconstruction program JULIA. As input we used the 461 event from the GALEPH run. The results are shown in Table 8.

Table 8: Total time for executing JULIA on all platforms
(Number of events = 461)

Platform	Total time (seconds)	Time ratio to VM	%	Normalised to 20 CU (%)	System utilisation
VM	1170	1.00	100	100	50%
CSF02	1042	0.89	112	83	71%
SAGA01	2227	1.91	53	58	91%
SHIFT9	1260	1.08	93	93	82%
RS-350	1972	1.68	60	120	99%

When we profiled JULIA, we naturally obtained quite a different distribution compared to what GALEPH has displayed:

Table 9: Time distribution in % for some of JULIA routines
(Number of events = 461; the symbol "-" means that the value is below threshold)

		SGI/150 SHIFT9		DEC/3400 SAGA08	
	Subroutine	% cycles	No. calls	% cycles	No. calls
JULIA	Bit-manip	11.4	900 000	4.5	227 000
	ufttkal	10.5	1235	5.4	1235
	ots_div	–	–	5.7	240 000
	tpkill	5.2	1	4.4	1
	tftwtb	4.7	24	3.1	24

There was a modest peak for the bit manipulation routines that were being used for packing and unpacking of data and bits-strings. We did not manage to determine the reason for the huge difference in the number of calls on SGI compared to DEC but it was suspected that the DEC compiler was better at inlining trivial calls. The second highest consumer, the helix fitting routine UFTTKAL, was being called for each track candidate and could not easily be simplified or optimised.

Given the low relative performance of DEC/OSF, we discussed the issue with one of the DEC systems engineers, Jean-Pierre Thibonnier [Ref. 5]. Very timely for both parties, a new DEC compiler (level 3.6) had just arrived and when using this version the results changed dramatically (from 48 to 83 or even 87% on DEC/3900).

Table 10: New timings for executing JULIA on selected platforms
(Number of events = 461)

Platform	CERN Unit	CU-ratio	Total time (s)	Normalised to 20 CU (%)	
AFAL11 (a)	52	2.6	968.	46.	
AFAL11 (b)	52	2.6	544.	83.	19/2/95
AFAL11 (c)	52	2.6	520.	87.	19/2/95
AFAL11 (d)	52	2.6	544.	83.	20/2/95
HP-735/125	34	1.7	802.	86.	
RS6K/590	27	1.35	839.	126.	

The first measurement (a) was done by using the same executable as SAGA01, i.e. no recompilation. The total time was 968 seconds which corresponds to user time (929.4) plus system time (38.3 seconds). The second measurement (b) was made by DEC using the O option (O4 level) of new FORTRAN Compiler release 3.6 combined with CERNLIB/94B. The third measurement used the same compiler and compiler options but linked in routines from CERNLIB/95A which had become available by then. An encouraging 4% improvement was obtained. The last measurement (d) confirmed that optimisation level (O2) gave the same performance improvement as O4 (in run b).

Given that all platforms were now at 83% or higher we did not pursue JULIA optimisation any further.

2.6 Analysis of ALPHA

The last task was to run the B-tagging analysis program, ALPHA. As input, 2000 events were

staged from tape. The results are shown in Table 12 under the heading Version 117. Execution times were relatively short compared to GALEPH and JULIA. It was nevertheless necessary to profile the program and try to understand the low performance on DEC and SGI; see Table 11.

Table 11: Time distribution in % for some of ALPHA routines
(Number of events = 2000)

		SGI/150 MHz SHIFT9	
	Subroutine	% cycle	No. calls
ALPHA	ggthru_	29.5	1 030
	shft_l	4.6	1 500 000
	findip_	4.6	187
	cnvcha_	2.3	3 550

The profile which was run only on the SGI showed a peak generated by the GGTHRU (THRUST) function. To save cycles in this case, we suggested to calculate this value once and for all for the whole collaboration and save it in banks or in a data-base. It should be recalculated only when a different kind of physics is needed.

In discussions with Aleph it was then discovered that by mistake O1 had been used as the optimisation level for the Aleph libraries on the two poorly performing machines. A new version, Alpha-118, was in the mean-time made available. When we ran the BT analysis with this version we got better, although not fully satisfactory results.

Table 12: Total time for executing the ALPHA on all platforms
(Number of events = 2000)

		Version 117		Version 118		Comments
Platform	Norm factor	time [sec]	Normalised performance	time [sec]	Normalised performance	
VM	1.0	123	1.00	126	1.0	
HP 735/99	1.35	128	0.81	118	0.79	
SGI/150	1.0	252	**0.49**	193	**0.65**	OPT O2
SAGA	0.9	348	**0.39**	196	**0.71**	OPT O2
RS-350	0.5	270	0.91	281	0.90	
RS-590	1.35	117	0.78	120	0.78	
HP 735/125	1.70	102	0.71	92	0.81	
DEC/3900	2.60	165	**0.29**	91	**0.53**	

At the end of the study the new DEC compiler (described in the previous section) was not yet the official version for producing the libraries in Aleph. It should definitely improve the performance on DEC/Alpha systems such as SAGA and the new DEC/3900 when that is done.

3. KERNEL STUDIES

3.1 Introduction

The previous examples with RNDM and IUHUNT were good demonstrations of the advantage of studying functions in an isolated manner. Wanting to build on success, we consequently felt that it was important to review other relevant "functions" that are frequently used in HEP programs. During the year we took such functions aside for a special analysis whenever discovered. The results are summarised in this chapter.

3.2 Bit shifting

One group of such analyses were related to the bit manipulation calls which were encountered in the JULIA and ALPHA programs. The high number of calls, 900,000 for JULIA on SGI, convinced us of its importance. To extract bits there are two functions that can be used, IBITS or ISHIFT, both of which are part of the MIL-STD (military standard). It turns out that IBITS can be replaced by ISHIFT according to the following equation:

IBITS(M, I, L) = ISHFT(ISHFT(M, 32-I-L), L-32)
where: 0 <=I <=31
1 <=L <=32 and I+L=32.

Normally manufacturers provide both functions via native implementations, but the CERN library also provide them, for the case where they do not exist. In our case, all vendors provided native versions. By comparing these through kernel tests, we got the following results:

Table 13: Summary of bit shifting tests:
(Number of calls is 1 million; timing in seconds)

Mode	CERN VM	HP-735 99	HP-735 125	SHIFT9 SGI/150	DEC-3400	DEC-3900	RS-350 IBM	RS-590 IBM
IBITS	17.7	**1.61**	**1.28**	9.45	**1.65**	**0.95**	17.15	**1.24**
ISHFT	**12.6**	4.76	3.74	**7.95**	3.28	1.63	**4.98**	2.28

On the RISC platforms, except on the RS6K-350 and the SGI, it was preferable to call IBITS. In term of absolute timing, the IBM RS6K-350 was doing rather badly, the RS-590 was much more acceptable. Additionally, it can be seen that CERNVM was actually a rather "bad" system for bit manipulations. It was clear from the whole exercise that there was very little to be gained by replacing IBITS calls with its "substitute" in a mechanical way. It would also have been a major effort to go through all the references in the programs and, made careful modifications.

As long as HEP programs are using packed data, however, functions like IBITS should always be scrutinised closely for abusive CPU consumption.

3.3 Bit testing

Another analysis came from Opal's reconstruction code (ROPE), which was using the BTEST(IA, I) function quite heavily. This FORTRAN function is used to test if a bit in position I of the word IA is set or not.

BTEST(IA, I) is equivalent to testing if IAND(IA, 2**I) is zero or not. Such tests occur

frequently, either in the original form of calling BTEST or in an IAND-like form, but we knew that using the power function (2**I) for masking purposes would be a very costly operation. We decided to make a three-way comparison, using the original BTEST, using the IAND-call plus the power function and using IAND with a pre-defined mask for the various 32 combinations. Table 14 shows the results where (a) corresponds to (IAND(IA, 2**I).ne.0) and (b) represents (IAND(IA, MASK(I)).NE.0).:

Table 14: Summary of bit testing tests:
(Number of calls is 1 million; timing in seconds)

Mode	CERN VM	HP-735 /99	HP-735 /125	SHIFT9 SGI/150	DEC-3400	DEC-3900	RS-350 IBM	RS-590 IBM
BTEST	9.44	1.78	1.40	5.28	**1.27**	**0.32**	**2.30**	**1.12**
(a)	10.94	19.12	15.08	13.02	15.42	8.42	19.57	2.09
(b)	**2.0**	**1.35**	**1.06**	**0.73**	1.32	0.65	3.06	1.32

As expected, the result shows that the version calling the power function should be avoided at all costs (although it may be "elegant" in the code). Between BMASK and IAND/IMASK the machines settled into two camps. DEC and IBM/RS6K were faster in the former case, HP, SGI and CERNVM in the latter. On CERNVM and SGI the difference was actually quite significant, making the SGI/Challenge the second best or the absolute worst RISC system depending on the chosen code. The DEC/3900 was the absolute fastest machine also in this case. In general we got the impression that DEC had optimised very well their library routines, so that when running on a fast machine, the results came out at the top (or close to it)

3.4 An analysis of DSIGN

When GEXAM1, the GEANT example 1, was profiled [See chapter 5] we saw a relatively high consumption inside the FTN_DSIGN function which is the double-precision FORTRAN SIGN function. Reducing the overhead which otherwise involves a jump to a function could be achieved by replacing the code with two FORTRAN statements since SIGN(a, b) = |a|*sign(b) translates into:

```
y = abs(a)
if (b.lt.0.) y = - y
```

The ABS (absolute value) function is often inlined by the compilers. Consequently we tested the following four combinations on the various platforms:

Case 1) DSIGN(a, b) when b>0
Case 2) DSIGN(a, b) when b<0
Case 3) The substitution code for b>0
Case 4) The substitution code for b<0

The results are shown in Table 15.

Table 15: Summary of DSIGN tests:
(Number of calls is 3 million; timing in seconds)

Case	CERN VM	HP-735 99	HP-735 125	SHIFT9 SGI/150	DEC-3400	DEC New compiler	RS-350 IBM	RS-590 IBM
1)	2.43	2.89	2.29	0.72	0.82	0.77	2.95	1.51
2)	2.51	2.92	2.31	0.75	0.77	0.67	2.99	1.50
3)	2.61	0.73	0.58	0.71	0.55	0.47	1.92	0.99
4)	2.67	0.82	0.65	0.78	0.73	0.62	2.05	1.03

The first two tests were slower on all platforms except on CERNVM and on SGI where the compiler seemed to take care of the inlining automatically. All the other systems saw a real gain when the manual inlining was applied. Using the latest version (3.21) of GEXAM1, however, there was no trace of DSIGN any more. The calls are still present in GSNGTR but they are no longer invoked in GEXAM1 in a significant way.

3.5 Measuring the whole set of arithmetic functions

Since tests of performance and accuracy of the FORTRAN functions has always been a method to verify the floating-point capability of "number- crunching" machines, it was only natural to port an existing test package , the UNFUN code [See Ref. 7] and modify it for our analysis. The package was enhanced to include all the known intrinsic FORTRAN functions as defined by the ANSI convention. In particular we added ATAN2 which is heavily used in HEP reconstruction codes. We ended up with 13 REAL*4 tests and the same number of REAL*8 tests. Each test is called with 500 random variables, chosen uniformly, from the full validity range of the functions. Each test is repeated 2048 times for the total measurements implying 1,024,000 calls to each function altogether. We went through several iterations of the tests until some hidden cycles of the UNIX subroutine calls were eliminated. The program was originally using an indirect function jump by passing the function name as an external to a general dispatching routine. The overall timing is given in Table 16 and the detailed results can be seen in Appendix V.

Table 16: Summary of Arithmetic Function tests:
(Number of calls is 1.024 millions; timing in seconds)

	CERNVM ES-9000	CSF HP-735 99 MHz	SHIFT9 SGI/150 150 MHz	SAGA01 DEC-3400 133 MHz	RS6K-590 IBM 67 MHz
Options:	OPT(3)	O2	O mips2	O2	O2, pwr1
Total time	23.43 s	26.77 s	33.37 s	35.08 s	41.68 s
Perf_ratio	1.0	0.88	0.70	0.67	0.56
Norm_ratio	1.0	0.64	0.70	0.74	0.42

By analysing the total time for each platform we concluded that all of them should have done better (compared to CERNVM) if the CERN Units ratios were used as an indication of the power of these machines. We see from the computed ratios that the efficiency of UNIX FORTRAN libraries is around 74% in the best case and about 42% in the worst case. The IBM-

RS-6000 system were particularly bad and this was mentioned to T.Bell/IBM. We were again rather pleased when a few months later a new version of the library lowered the total time from 41.68 seconds to 20.18 seconds (which is more than a factor 2 faster).

Clearly such tests are quite sensitive to the usage of machine features linked to compiler options that enhance the performance of a particular function. This is, for instance, the case with the mips2 option on SGI which enables a hardware square-root instruction. On the other hand if a bad algorithm is used in the library, no optimisation techniques whatsoever can help. This was exemplified by the Atan2 function in the original IBM library. Fortunately the improvement in the second version was quite significant.

4. FREQUENTLY USED CERNLIB ROUTINES

We knew that some of the KERNLIB routines had been written in assembly code on CERNVM, whereas on the RISC platforms they were kept in FORTRAN. Consequently, we selected for our analysis a few routines that are used quite frequently in many applications, e.g.: UCOPY, UZERO, VZERO [See Ref. 4]. These routines consumed approximately 3-5% of total CPU time on the RISC platforms when running the various Aleph programs. By comparing runs with either the library routines or with sources included in the FORTRAN compilation we were able to judge the impact of the assembler versions:

Table 16: Summary of selected KERNLIB tests:
(Size of array is 299; timing in seconds)

System	VM(600J)		HP-735/99		SGI/150		DEC/3400		IBM/RS-390	
Routine	library	fortran	library	fortran	library	fortran	library	fortran	library	fortran
ucopy	7.8	26.9	9.4	9.4	6.9	6.8	6.6	16.1	9.2	9.1
uzero	3.5	23.3	6.2	6.2	4.3	4.3	4.4	13.8	8.9	9.3
vzero	3.3	23.3	6.2	6.2	4.3	4.3	4.4	11.6	9.1	9.1

As expected we observed that the library versions of CERNVM were much more efficient in all three cases. A big surprise came from the observation that the DEC tests also indicated that the library routines were much faster. The answer to the puzzle came from playing with optimisation levels. When using O4, the default, as opposed to O2, our "safe" choice we managed to reproduce the library values with our inlined source. CERNLIB had simply been generated with the default option (O4). On all the other platforms we got practically identical results. Trying to run the GALEPH code with the FORTRAN version on VM slowed the program down by only about 1%. It was therefore not considered to be a serious issue.

As we will see later, however, the issue became more serious in the case of some of the jobs of Atlas.

5. REALISTIC AND OPTIMISTIC BENCHMARKING

During the study it was necessary to deploy E.McIntosh's benchmark suite for three main reasons:

• We did not have the latest CERN Unit values for the most recent RISC platforms introduced at CERN. This was the case for the DEC-ALPHA 3900, the new SGI/Challenge (200 MHz), and the IBM/RS-590 machine.

• Several of the benchmarks measurements had been made a few years ago (See Ref. 2), and since then compilers and libraries had (most likely) improved so it was important to get updated

values.

• Since the expected performance (of real jobs) is so strongly linked to these "sacred" CERN Units numbers, it was necessary to understand under what conditions they had been measured. System load conditions, stand alone runs as well as a multitude of compiler options could have had large impacts on the results. Finally it was necessary to see the impact of "realistic" options corresponding to what a normal experiment would use. In other words, how stable were the CERN Units and how big would the error bars be ?

The main part of the benchmark contains four jobs: CRN3, CRN4, CRN5 and CRN12. CRN3 is an old FOWL Monte Carlo test. CRN4 is a "modern" LUND Monte Carlo generator. CRN5 is GABI, an old event processing program (with 2 input files). CRN12 is JAN, an old event processing program (also with 2 input files).

These four tests are used to calculate the CERN Unit by obtaining the ratio of the CPU timing for each test with known numbers from the VAX-8600 which had been defined equal to one CERN Unit. From the 4 ratios, the geometric mean is calculated to give the "official" value. (See Ref. 2 for further details)

The other parts of the benchmark suite are used for compiler testing, vectorisation testing and are not included in the calculation of the CERN Unit.

The results obtained are summarised in Table 18:

Table 18: Summary of selected benchmark runs:
(Measured CERN Units)

System	HP/735/99	DEC/3400	SGI/Challenge 150 MHz	IBM RS6K/590
Optimistic value (CU)	28.1	18.0	22.1	27.1
Realistic value (CU)	23.5	16.0	20.9	21.4

The full set of tests can be inspected in Appendix VI. There is one set of numbers for the "realistic" conditions where we use a reasonable optimisation level (O2), the appropriate hardware switches (i.e. mips2 or pwr2) and static (or K) when it helped. In the “optimistic” runs we used the highest optimisation levels, fast libraries if they existed, and the same hardware switches as in the first case. Under these conditions we evidently got numbers that were close to the values distributed by the vendors. These values should consequently be considered as the “ultimate” performance peak, but in practice, regular HEP jobs do not reach these values because the optimisation levels are more modest (to be sure that reliable numbers are produced). The same argument goes for fast (but inaccurate) libraries where a certain number of “rounding errors” can be expected.

Clearly all vendors want to "show off" and present the maximum performance they can extract from their machines. That is fine. In our case it helped explain where some of the missing Aleph performance was hiding. The full table shows that some platforms lose between 10% and 30% of their CERN units due to this phenomenon.

Another aspect that we could observe, were the fluctuations of the CERN Units as a function of the machine load. To minimise errors, each test should be run three times to get the average (or force a review the run conditions if the results vary too widely). In addition CRN4 should be run using 10,000 events and not only 1000 in order to reduce the sensitivity of these fluctuations. The running time for 1000 events is nowadays only a few seconds on most machines so a factor of 10 will improve stability of the results Additionally, it will give all four tests almost equal weight in the geometric mean calculation.

6. GEXAM1 MEASUREMENTS

We decided to run GEXAM1[12] on various platforms in order to study some of the points that were found in the GALEPH code, like the importance of different algorithms for random numbers, and to measure yet another set of performance ratios. GEXAM1 is also heavily used by the computer vendors to demonstrate the "power" of new platforms just like the "official" benchmarks. For these reasons it was useful to compare performance as influenced by various compiler options as well as machine load.

The first check was to see if the Random Generator (GRNDM) consumed large amounts of CPU cycles or not. When replacing GRNDM with our RANLUX version there was very little improvement (5-6% on VM and 2-3% on HP) as can be seen from the two tables in Appendix VII. The reasons were found to be twofold:

• The relatively low number of calls that were being made (only 50,000 calls/event are produced in Gexam1 whereas in GALEPH there are 2.7 million calls/event).

• The GRNDM version used by GEXAM1 is the REAL*8 version and not the integer version of the same algorithm (RANECU). Consequently, the measurement showed a small total consumption, even on HP systems.

The second check was made with the integer version of RANECU that had been suggested by HP (see Section 2.2). When running GEXAM1 on a HP-735/99 replacing the original GRNDM with the HP version it ran in 1.574 seconds/event (on a 100 event sample), compared to original GRNDM run of 1.498 seconds/event. We concluded that there was no easy way to improve the REAL*8 algorithm.

Problems arose, however, when the random seeds were compared and different random values started appearing in the first events. This led to a series of checks between the two versions using stand-alone versions of GRNDM to compare the last seeds but even after 5 million iterations the last seeds were still the same. When we checked carefully the last random values we noticed differences in last digits which occurred due to the different way they were being calculated in GRNDM :

In the HP version:

value = REAL(iz)*4.6566128e-10 equal to: r*4=(r*4)*(r*4)

In the original version:

value = iz*4.6566128e-10 equal to r*4=(r*8)*(r*4)

In other words the conversion of iz was different in the two cases and this led to one or two decimal digits of difference which in turn forced the Monte-Carlo program into a different path. This was therefore a good example of the problem that minor inaccuracies can lead to the generation of different events. As a result we activated a more relevant comparison of the GEXAM1 results by analysing the histograms. Fortunately they agreed within a 1 or 2% for all the distributions.

During the year we were able to run two versions of GEXAM1:

• The old GEANT 3.15 version (used by many vendors) as a stand-alone source package that does not require the CERNLIBs because all the source files are available.

• The latest GEANT 3.21 version which was using the standard libraries on all the platforms as well as an input data file.

The results show that version 3.21 consumes at least 20% more time than version 3.15 in this case. This is mainly due to the added code for photon physics and the detector geometry in GEXAM1 which is so trivial that the virtually divided volume feature in 3.21 gives no benefit in contrast to more complex detectors like the ones used in LEP or LHC[13].

---

[12]The source of GEXAM1 (3.15) was taken from the “hptrip” source directory of R.Brun. A later verification showed that timing results are the same if CERNLIB with GEANT315 is used instead.

[13]Admittedly this is a problem and an effort is now underway to try to understand if other GEXAM examples (such as GEXAM4) would reflect the simulation of real detectors more convincingly.

We tried on a few occasions to check the effect of higher optimisation levels and the architecture dependent features (i.e. -mips2 or -pwr2) on the relevant platforms. There was some improvement (6-7%) when pwr2 was used on the IBM/SP2 which in fact provides hardware instructions for certain functions (i.e. sqrt, exp, log). No improvement was seen, however, when the highest optimisation and automatic inlining capabilities were used (-O3, -Q, -qarch=pwr2). From this fact, it was concluded that automatic inlining is not always a recipe for improving performance. Inlining must be done in a selective way by running a profiler on the code. Since GEXAM1 profiling did not show any "interesting" peaks in its distribution, we doubted its usefulness. Using optimisation level O4 instead of O2 on HP confirmed this doubt by giving worse performance. The next thing was to check code instrumentation on the same platform. This did actually give an extra 5% improvement, but not more (See Appendix VII for details).

7. PROFILING OF ATLAS CODE

7.1 Introduction

Full of “wisdom” from the Aleph experience we decided to use the same tools and techniques for investigating the programs of one of the LHC collaborations, namely Atlas. Beyond discoveries of performance "bugs", we were also hoping that they would be rather interested in having an ongoing discussion about performance issues in general since they were still redesigning and enhancing a lot of their codes.

7.2 The Atlas simulation program (DICE)

First we looked at the ATLAS Monte-Carlo programme, DICE, on the HP platforms in use by the collaboration. DICE is currently based on GEANT 3.21 so that it can profit from the virtually divided volume feature. In the first tests, we used data records to send single muon particles into the detector. These jobs took 210 seconds/event on a HP/735/99.

Since we were already well versed in random number issues we started by reviewing the RNDM and GRNDM calls. The results (for 3 events) showed 82 calls to RNDM and 4.7 millions to GRNDM with an average of 2.7 numbers per call or 4.3 millions random variables per event. Running with the RANLUX version we significantly reduced the timing to only 70 seconds per event (a factor 3), but since different events came out due to a different random walk, we needed to check it on full physics events which are more representative and take at least 5-6 minutes per event depending on the multiplicity. Higgs events were being used.

When we profiled such DICE runs without or with RANLUX we were able to improve the overall performance by 10% on HP platforms (see Table 15 in Appendix IX and the accompanying comments). Other improvements might be possible but they would not contribute too much since the relative time consumption of the other routines was small. A large amount of time was spent in GTNEXT of GEANT and that routine plus the other tracking routines were the major time consumers (six routines absorbed 32.1% or 37.1% respectively). We doubted, however, that anything could be done to improve them since the main objective of Geant version 3.21 which they were already using, had been exactly that.

Since we were impressed by the large number of subroutine calls observed in the profile, we tried to come up with an estimate of total subroutine calls per full event. The multiplication factor in brackets represents the number of estimated calls inside the quoted routine. For a single "long" event, which took 2240 seconds on a HP/712, we obtained:

• GTNEXT	12.8 M (* 5)	64 M calls
• GTRACK	1.6 M (* 10 in stepping)	16 M calls
• XTRDGEN	1.6 M (* 4)	6 M calls
• GRNDM	8.0 M	8 M calls
Sub Total:	In Geant alone:	100 M calls
+ all other FORTRAN calls, ZEBRA, system, etc.		100 M calls
Grand total:		200 M calls

From earlier studies we had estimated that a jump to a subroutine with one argument would take 0.2 micro-seconds per call on a HP/735/99 machine (about 20 instructions). Since the average number of arguments per call is larger than one we rounded it up to 0.3 micro-seconds for a slightly slower HP/712. Total time spent in subroutine calls for an event was then 200 millions * 0.3 microseconds or 60 seconds which amounted to 2.7% = 60/2240 of total time. This might be an underestimate but we concluded that it is in any case not the major CPU consuming portion of DICE. Random number generation (which we improved with RANLUX) and particle tracking (even with virtually divided volumes) remained the two biggest CPU consumers in DICE.

7.3. Running ATRECON (the Atlas reconstruction program)

We ran ATRECON by taking the input from one of Atlas's official tapes, I28445. The results were compared with an existing Atlas output for correctness. For 80 triggers we got an average timing of 1.88 seconds per event, for 120 triggers we got 1.46 seconds.

When we profiled the executable for ATRECON (using GPROF) on 100 events we obtained the following results:

Table 19: Time distribution in % for the main ATRECON routines
(Number of events = 100)

time in %	cumulative seconds	seconds	name
53.6	92.70	92.70	vzero_
16.1	120.52	27.81	hadcomb_
8.7	135.51	15.00	xtrddig_
4.6	143.40	7.88	cemcand_
2.4	147.51	4.12	xfinter_
1.8	150.66	3.15	hadcand_
1.4	153.03	2.37	hadunpk_

It was clear from these measurements that VZERO was being called an incredible number of times. In order to find where it was being called, we added the source of the subroutine to the file compiled with -G option and let GPROF produce the relevant statistics of these calls. It was also interesting to discover the vector length which was being used most often on these calls. A histogram showed that 26.5 millions calls were being made with a vector length below 500 (out of 26.6 millions). A more accurate run showed that the average lay between 125 and 149. GPROF pointed out that subroutine XTRDDIG was making 25.45 millions calls to VZERO. The source of this routine showed three different calls with three different vector length: 20, 80, and 128. It was clear that it had to be the last one. It turned out that in the code of XTRDDIG there was a triple loop which had the call buried in the beginning of the innermost loop. Looking more carefully into this loop, one could see the following strange situation:

inner do loop....
k=idep(loop index)
call vzero(uuu, 128)
if (k.eq.0) go to end of loop.....

It was clear that one should move the call beyond the If statement. By doing so the number of calls dropped to 41,000 instead of millions and the most important thing was that the time per event dropped by almost 50%. (and the results were identical). The time per event dropped from 1.46 to 0.834.

Checking if an inlined unrolling of VZERO could improve more the timing per event but we only got a 0.3% improvement. We checked to see if other routines might help save few CPU cycles but we could not find other candidates. The ATLAS software group has been notified about this unfortunate “bug” and by now a correction should have been introduced.

8. CONCLUSIONS AND RECOMMENDATIONS

As far as the missing CERN Units in ALEPH are concerned, we believe that we found the answers why CERNVM appeared so efficient:

- Routines in GALEPH (RNDM, IUHUNT) suffered from lack of efficiency and needed repair.
- JULIA and ALPHA suffered from an early version of the DEC’s new RISC compiler.
- Some efficiency was lost in the CERNLIB kernel routines.
- The CERN Units quoted by manufacturers were higher than what we could get with “realistic” compiler options, i.e. the ones an experiment tend to use.

From all the various investigations we tried to draw some general conclusions. It is our hope that these recommendations will help create the "performance awareness" that was discussed in Chapter 1. As already stated, any performance action must be undertaken by keeping in mind the other costs, such as manpower spent to provide an improvement. Our conclusions are listed in some kind of "priority" order:

• Do the design with performance in mind:

Nothing beats a good design where the performance implications have been taken into account right from the beginning. This means, in our opinion, that the developers need to have good ideas of where the "hot spots" (normally the inner loops, but not only) will be when the program is complete. It may be quite challenging to get everything right in cases where the application is being used in many different contexts. This would typically be the case for a framework like GEANT where the "user" determines the actual parameters of the detector (number of volumes, relationship between volumes, etc.) and also the kind of physics that will be simulated.

• Repair the design with new code.

For very good reasons an initial design (with its performance estimates) may not be valid for ever. A linear search of short queue search may be a valid thing to do in a non-critical part of an application, but a rewrite to use a binary tree search may be needed when the application grows beyond the first usage patterns. For people with good memories, the Wylbur keyword (user/account) lookup mechanism and the need to rewrite it at the heights of CERN’s Wylbur usage may spring to mind as a relevant example. Rewrites therefore become a common way of enhancing a program through its life cycle. The introduction of virtual volumes in GEANT 3.21 is such a rewrite and it belongs to this report to say that Aleph rewrote the part of GALEPH that used IUHUNT (for simulating the noise of the Vertex detector) so that the bad usage was eliminated altogether.

• Profile all your programs regularly.

The way to identify a hot spot that needs a rewrite consists most often of a profiling exercise. In many cases, a major rewrite may not be necessary and only minor surgery may suffice. Our recommendations to use RANLUX in GALEPH and to move VZERO inside XTRDDIG in ATRECON belong to this category. In any case, it is strongly recommended to profile major physics programs regularly, either for each major release or a certain number of times per year. A port to a new hardware/software platform also warrants a profiling exercise, because, as we demonstrated with RNDM in GALEPH, a minor hot-spot on one platform may all of a sudden become a major bottleneck on another. We leave it to the right competent body (such as the GEANT co-ordination meeting) to discuss which random number generation routine should be used by default in GEANT. RANECU is probably in use because for historical reasons, but somebody may need to do a fair bit of analysis of good randomness and long periodicity to convince the whole HEP community that another one, such as RANLUX or RM48, is better. Interestingly enough, we noticed "en passant" that GISMO is using RANLUX at luxury level 2.

• Optimise "hot-spot" candidates in the CERN library.

In the past, we used to rewrite the likely candidates for hot-spot creation in assembly. This was done on CERNVM and on VMS/VAX. With the advent of RISC computers (Relegate the Important Stuff to the Compiler) the decision was made to keep even all the kernel routines (in KERNLIB) in a high-level language (C or FORTRAN). Our recommendation is not to reverse the trend (to higher-level programming and easier maintenance) but we do recommend that users are made aware of likely candidates for hot-spot creation. Maybe a tuning guide, or tuning examples in the CERN Computer Newsletter will help. Maybe this very report will be of some help to the user community.

• Understand the performance potential of the platform in use.

This is quite an important point and should maybe be higher in the list. During our study we demonstrated the importance of understanding and exploiting compiler options that impact performance. Moving to higher compiler options must always be followed by a verification of the correctness of the results. We also showed the need to upgrade to new compiler versions (as DEC demonstrated with JULIA) whenever major new versions appear. With IBM we saw the need to feed back to the manufacturer non-competitive results from the use of their libraries.

• A good knowledge of which statements generate ineffective code.

The examples of DSIGN, BTEST and others served to show that often good knowledge of what becomes efficient or inefficient code can create a significant advantage in terms of efficiency. A compiler may not always be able to "smooth over" such coding examples with the result that performance penalties are collected during execution. One simply cannot under-estimate the importance of providing coding examples as well as DOs and DONTs to the users. Further reading [such as Ref. 6 and 8] is highly recommended.

REFERENCES

[1] Alpha job production on Shift9 , Eric Lançon, CEA/DSM/DAPNIA/SPP/107099/000
[2] Benchmarking Computers for HEP, Eric McIntosh, CN/92/13
[3] Bob Montgomery, (HP USA) communications by E-Mail to S.Jarp
[4] Routines, functions or Programss taken from the CERNLIB packages.
[5] Jean Pierre Thibonnier from the DEC Joint Project Office at CERN
[6] High performance Computing, Kevin Dowd, O'Reilly & Associates.
[7] Package was taken from the 370/E Emulator diagnostics package. Written by R.Yaari (Weizmann Institute, 1983-87).
[8] "FORTRAN Optimisation" Michael Metcalf, Academic Press, 1982, 1985.

APPENDICES

Appendix I) Machine specifications: MHz, SPECint92, Official CERN Units per platform

Table 1: Frequency, CERN-units, SPECint92, SI/CU ratio, normalisation factor for each platform (plus the name of the most frequently used machine in each category).

Platforms	MHZ	CERN units	Spec-int92	ratio Specint/C.U	Norm factor f=cu/20 (vm)	CERN name
IBM/ES9000	--	20	---	---	1.0	CERNVM
HP-712	80	20.0	84	4.1	1.05	CSF45
HP-735	99	27.0	109	4.0	1.35	CSF01-25
HP-735	125	34.1	136	4.0	1.70	Shift21
DEC-3400	133	18.0	75	4.1	0.90	Saga01-08
DEC-3900	275	52.0	189	3.9	2.60	Afal11
SGI-Chall	150	20.0	90	4.5	1.00	Shift9
SGI-Chall	200	29.0	119	4.1	1.45	Shift7
IBM-RS/590	67	27.0	121	4.5	1.35	Rsibm3
IBM-RS/350	47	10.0	35	3.6	0.50	Rsalmp01

Appendix II) GRNDM study

As a related issue to the RNDM study described in section 2.2, we looked at the effectiveness of fetching multiple random numbers in each call.

Table 2: GRNDM runs, comparing different fetches (1-10) using original GRNDM version. All opt-levels were O2

Number of fetches	VM	SHIFT9	SAGA01		CSF02	RS-590
				new compiler		
1	3.18	4.68	6.78	3.13	7.62	2.25
2	2.48	4.38	6.67	2.90	7.12	1.92
3	2.23	4.25	6.82	2.85	6.97	1.83
4	2.10	4.21	6.72	2.83	6.89	1.78
5	2.00	4.13	6.45	2.75	6.77	1.71
10	1.90	4.11	6.43	2.78	6.70	1.69

The more is fetched in one go, the less the time per variable becomes. The time seemed to converge slowly, around 10 fetches, on all platforms. We reran the same tests using the RANLUX algorithm instead of the original one.

Table 3: GRNDM runs, comparing different fetches (1-10) using RANLUX within GRNDM. All opt-levels were O2

Number of fetches	VM	SHIFT9	SAGA01		CSF02	RS-590|
				new compiler		
1	2.00	1.16	2.40	1.18	1.62	1.51
2	1.32	0.96	2.25	1.00	1.15	1.30
3	1.05	0.89	2.10	0.90	0.98	1.22
4	0.94	0.84	2.02	0.87	0.87	1.16
5	0.83	0.75	1.75	0.80	0.79	1.08
10	0.74	0.76	1.85	0.85	0.75	1.09

The same increased efficiency was observed. It was also noted that the RANLUX version is always much faster than the original one.

Apendix III) Source code of (key portions of) the random number generators

RNDM algorithms.

RNDM original Algorithm (as used by Aleph)

```
C =======Replacing the call to RANECU(RVEC, LRVEC)===============
      DO 100 I= 1,  LRVEC
      K = ISEED1/53668
      ISEED1 = 40014*(ISEED1 - K*53668) - K*12211
      IF (ISEED1. LT. 0) ISEED1=ISEED1+2147483563
C
      K = ISEED2/52774
      ISEED2 = 40692*(ISEED2 - K*52774) - K* 3791
      IF (ISEED2. LT. 0) ISEED2=ISEED2+2147483399
C
      IZ = ISEED1 - ISEED2
      IF (IZ. LT. 1) IZ = IZ + 2147483562
C
      RVEC(I) = REAL(IZ) * 4.6566128E-10
 100  CONTINUE
C ================= end of RANECU ==============================
```

RANLUX Algorithm (level=0):

Initialization:

```
     DO 25 I= 1,  24
     TWOM24 = TWOM24 * 0.5
     K = JSEED/53668
     JSEED = 40014*(JSEED-K*53668) -K*12211
     IF (JSEED. LT. 0) JSEED = JSEED+ICONS
     ISEEDS(I) = MOD(JSEED, ITWO24)
  25 CONTINUE
```

Real Routine :

```
C ====REPLACE THE RANECU ALGORITHM WITH RANLUX VERSION FOR UNIX==
   DO 100 IVEC= 1,  LRVEC
   UNI = SEEDS(J24) - SEEDS(I24) - CARRY
   IF (UNI. LT. 0.) THEN
     UNI = UNI + 1.0
     CARRY = TWOM24
   ELSE
     CARRY = 0.
   ENDIF
   SEEDS(I24) = UNI
   I24 = NEXT(I24)
   J24 = NEXT(J24)
   RVEC(IVEC) = UNI
C small numbers with less than 12 "significant" bits are "padded".
   IF (UNI. LT. TWOM12) THEN
     RVEC(IVEC) = RVEC(IVEC) + TWOM24*SEEDS(J24)
C    and zero is forbidden in case someone takes a logarithm
     IF (RVEC(IVEC). EQ. 0.) RVEC(IVEC) = TWOM24*TWOM24
   ENDIF
 100 CONTINUE
 ================== END OF RANLUX ================================
```

Appendix IV) Source code of IUHUNT

```
   FUNCTION IUHUNT (IT, IA, N, NW)
C
C CERN PROGLIB# V304  IUHUNT     . VERSION KERNFOR 2.09 751101
C ORIG. 01/09/72
C
   INTEGER IT, IA
   DIMENSION IA(9)
   IF (N.EQ.0) GO TO 19
   IF (NW.EQ.0) GO TO 20
   DO 18 J=1, N, NW
   IF (IA(J).EQ.IT) GO TO 21
  18 CONTINUE
  19 IUHUNT= 0
   RETURN
  20 IF (IT.NE.IA(1)) GO TO 19
   IUHUNT= 1
   RETURN
  21 IUHUNT= J
   RETURN
   END
```

The reason why the UNROLLED version looks "messy" is the fact that we had to keep a different version for each increment.

```
   FUNCTION IUHUNC (IT, IA, N, NW)
C
C CERN PROGLIB# V304  IUHUNT     . VERSION KERNFOR 2.09 751101
```

```
C ORIG. 01/09/72
C MODIFIED TO BE UNROLLED FOR NW=2 OR 3 (CORESPONDS TO ALEPH
CHOICE)
   INTEGER IT, IA
   DIMENSION IA(9)
C
   IF (N.EQ.0) GO TO 19
   IF (NW.EQ.0) GO TO 20
     JJ=MOD(N, 4*NW)
     L=(JJ-1)/NW
     J1=(L+1)*NW+1
     IF(JJ.EQ.0)J1=1
    IF(NW.EQ.2)THEN
CRY--- PREPARE THE PRECONDITIONING LOOP --------
C    JJ=MOD(N,  8)
    DO 82 J=1, JJ, 2
     IF (IA(J).EQ.IT) GO TO 30
  82  CONTINUE
C    J1=JJ+1
C    IF(MOD(JJ, 2).EQ.1)J1=J1+1
    DO 182 J=J1,  N, 8
     IF (IA(J).EQ.IT) GO TO 30
     IF (IA(J+2).EQ.IT) GO TO 32
     IF (IA(J+4).EQ.IT) GO TO 34
     IF (IA(J+6).EQ.IT) GO TO 36
 182  CONTINUE
     IUHUNC= 0
     RETURN
    ELSEIF(NW.EQ.3)THEN
CRY--- PREPARE THE PRECONDITIONING LOOP --------
C    JJ=MOD(N, 12)
    DO 83 J=1, JJ, 3
     IF (IA(J).EQ.IT) GO TO 30
  83  CONTINUE
C    J1=JJ+1
C    IF(MOD(JJ, 2).EQ.0)J1=J1+1
    DO 183 J=J1, N, 12
     IF (IA(J).EQ.IT) GO TO 30
     IF (IA(J+3).EQ.IT) GO TO 33
     IF (IA(J+6).EQ.IT) GO TO 36
     IF (IA(J+9).EQ.IT) GO TO 39
 183  CONTINUE
   IUHUNC= 0
   RETURN
    ELSEIF(NW.EQ.1)THEN
    DO 81 J=1, JJ
     IF (IA(J).EQ.IT) GO TO 30
  81  CONTINUE
   DO 181 J=1+JJ, N, 4
   IF (IA(J).EQ.IT) GO TO  30
   IF (IA(J+1).EQ.IT) GO TO 31
   IF (IA(J+2).EQ.IT) GO TO 32
   IF (IA(J+3).EQ.IT) GO TO 33
 181 CONTINUE
```

```
      IUHUNC= 0
      RETURN
       ELSE
       DO 18 J=1, N, NW
        IF (IA(J).EQ.IT) GO TO 30
     18  CONTINUE
      ENDIF
     19 IUHUNC= 0
      RETURN
     20 IF (IT.NE.IA(1)) GO TO 19
      IUHUNC= 1
      RETURN
C
     30 IUHUNC= J
      RETURN
     31 IUHUNC= J+1
      RETURN
     32 IUHUNC= J+2
      RETURN
     33 IUHUNC= J+3
      RETURN
     34 IUHUNC= J+4
      RETURN
     36 IUHUNC= J+6
      RETURN
     39 IUHUNC= J+9
      RETURN
      END
```

Appendix V) Kernel tables

Table 4 Arithmetic function timings -Direct measurements (2048x500) using unfun4.f. January 1995 (Timings in sec). Platforms of Group I:

Function	CERNVM IBM-ES 9000 OPT(3)	CSF HP-735 (99 MHz) O2	SHIFT9 SGI/Chall (150 MHz) O, mips2	Saga01 DEC-3400 (133 MHz) O2	RS6k-590 IBM (67 MHz) O2, pwr1
SQRT	0.37	.12	0.36	1.02	.73
DSQRT	0.45	.18	0.66	0.98	.71
EXP	0.52	.87	1.06	1.05	.91
DEXP	1.10	.92	1.27	1.25	.81
LOG	0.69	.91	1.21	0.85	.91
DLOG	0.87	1.04	1.27	1.05	.85
COS	0.79	1.04	1.08	1.13	.51
DCOS	0.90	1.06	1.05	1.25	.53
SIN	0.76	1.01	0.77	1.10	.51
DSIN	0.84	1.11	1.01	1.43	.49
TAN	0.96	1.30	0.90	1.30	1.06
DTAN	1.27	1.41	1.51	1.95	1.05
ACOS	0.70	1.15	1.39	1.15	.96
DACOS	0.85	1.30	2.07	1.88	.95
ASIN	0.71	1.16	1.38	1.22	.87
DASIN	0.83	1.29	2.05	2.10	.76
ATAN	1.18	.90	1.33	1.08	.88
DATAN	1.32	1.00	1.85	1.35	.88
ATAN2	0.68	1.24	1.33	2.07	**5.91**
DATAN2	0.92	1.42	2.14	2.82	**5.87**
SINH	0.59	1.03	0.86	0.83	**2.71**
DSINH	1.20	.76	1.02	1.08	**2.76**
COSH	0.62	1.04	1.44	0.87	**2.05**
DCOSH	1.77	1.35	1.78	1.45	**2.00**
TANH	0.75	1.01	1.16	1.35	**3.01**
DTANH	1.07	1.15	1.42	1.47	**3.00**
Total time	23.43	26.77	33.37	35.08	41.68

Table 5 Relative ratios of total arithmetic functions (compared to IBM ES/9000)

	CERNVM	CSF	SHIFT9	Saga01	RS6k-590
Time ratio	1.0	0.88	0.70	0.67	0.56
Norm. ratio	100 %	65 %	70 %	74 %	42 %

Observations:

The HP-735/99 was the fastest RISC platform in these tests. It performed hardwired square roots, and very efficient exponentials and log functions.

The IBM/RS6K-590 was not showing its potential. When we looked at the last 8 functions, we noticed the worst performance of all the platforms. We "complained" to IBM and the improvement came four months later, (see Table 6 in the Appendix).

Our results bore no relationship to CERN Units ranking. In other words, the internal algorithms decide much more than raw speed.

Table 6 Arithmetic fuctions timings -Direct measurements (2048x500). Using UNFUN4.f, January 1995 (Timings in seconds). New platforms

Function	CERNVM IBM-ES 9000 OPT(3)	CSF HP-735 (125 MHz) O2	SHIFT9 SGI (200b MHz) O mips2	AFAL11 DEC 3900 (275 HMz) O2
SQRT	0.37	.09	0.27	0.48
DSQRT	0.45	.15	0.50	0.45
EXP	0.52	.68	0.79	0.50
DEXP	1.10	.73	0.95	0.60
LOG	0.69	.72	0.91	0.38
DLOG	0.87	.83	0.95	0.55
COS	0.79	.82	0.82	0.53
DCOS	0.90	. 83	0.79	0.60
SIN	0.76	.84	0.57	0.53
DSIN	0.84	.85	0.75	0.58
TAN	0.96	1.50	0.68	0.58
DTAN	1.27	1.51	1.12	0.70
ACOS	0.70	1.05	1.04	0.55
DACOS	0.85	1.05	1.55	0.73
ASIN	0.71	1.02	1.03	0.57
DASIN	0.83	1.04	2.01	0.72
ATAN	1.18	.79	0.99	0.48
DATAN	1.32	.79	1.39	0.53
ATAN2	0.68	1.14	0.99	0.83
DATAN2	0.92	1.13	1.61	0.92
SINH	0.59	.81	0.64	0.40
DSINH	1.20	.61	0.76	0.48
COSH	0.62	.82	1.08	0.42
DCOSH	1.77	1.08	1.32	0.52
TANH	0.75	.80	0.84	0.50
DTANH	1.07	.91	1.06	0.62
Total time	23.43	22.59	25.41	14.77

Table 7 Relative ratios of total arithmetic functions (compared to IBM ES/9000)

	CERNVM	CSF	SHIFT9	AFAL11
Time ratio	1.0	1.04	0.92	1.59
Norm. ratio	100 %	61 %	64 %	61 %

Observations:

The DEC-3900 (275 MHz) was the fastest platform for these tests. It was 59 % faster than the ES-9000. Although these new machines run faster, they actually worsen compared to CERNVM when we use the normalisation ratio based on CERN Units.

Table 8 Updated results of Function tests on RS6K-590, using the latest IBM Fortran libraries. Two sets of compiler options were being used (pwr1 and pwr2) [Only functions with improved timing are listed]

Library used:	/usr/lib/libmass.a		/usr/lib/libm.a	
Date:	Apr 5 1995		Jan 30 1995	
Function:	O2 pwr1	O2 pwr2	O2 pwr1	O2 pwr2
SQRT	.53	.43	.73	.44
DSQRT	.49	.45	.71	.43
EXP	.33	.33	.91	.88
DEXP	.32	.32	.81	.82
LOG	.58	.58	.91	.91
DLOG	.54	.54	.85	.86
DLOG	.28	.28	.51	.52
DCOS	.28	.28	.53	.50
SIN	.28	.28	.51	.50
DSIN	.28	.28	.49	.50
TAN	.61	.61	1.06	1.04
DTAN	.58	.59	1.05	1.05
ATAN	.78	.78	.88	.89
DATAN	.75	.74	.88	.83
ATAN2	1.13	1.14 <<<	5.91	6.00
DATAN2	1.11	1.12 <<<	5.87	6.01
SINH	.53	.52	2.71	2.75
DSINH	.49	.50	2.76	2.81
COSH	.42	.42	2.05	2.17
DCOSH	.42	.42	2.00	2.07
Total time	20.34	20.18	41.68	41.68

Observations:

The new library was faster by a factor 2.05 when "pwr1" was used, and a factor 2.07 when "pwr2" was used. The usage of the latter option did not show a significant improvement maybe

explained by the fact that the library code is the identical in both cases.

The major change was in the D/ATAN2 functions which had been improved by a factor 5.2. There was no change in performance of TANH but this function is not frequently used in HEP code and if someone needs them, (s)he can use the ratio of SINH/COSH which should be faster. The RS6K-590 now obtained a much better effeciency factor: 23.43/20.43/1.35= 0.85 (compared to 0.42 before the library change).

Appendix VI) Benchmark tables:

Table 9. CERN Units for Unix Platforms
Realistic values vs Optimistic (under optimum conditions).
[Measurements made using CERN's official Benchmarks package]

		CERN Units		"Realistic" options	CERN Units		"Optimistic" options	"Official"
Platforms	MHz	GM	Range	used	GM	Range	used	CU
HP-750	66	12.1	9.3-16.4	-O, -K, -static	13.4	10.9-16.9	+O3, -K	10
HP-712	80	18.1	12.2-26.2	+O2, -K	19.9	12.9-27.3	+O4	n.a
HP-735	99	23.5	16.9-32.5	+O2, -K	28.1	18.1-38.8	+O4	27
HP-735	125	32.0	22.3-45.9	+O2, -K	35.4	23.3-51.2	+O4	4
DEC-3400	133	16.0	13.2-19.2	-O2, -static	18.0	14.4-23.1	-O4, fast	20
DEC-3900	275	37.9	32.7-44.0	-O2, -static	43.9	37.4-53.1	O4, fast	52
SGI-Challenge	150	20.9	17.1-24.6	-O2, mips2, -shared	22.1	18.1-25.2	*[14]	20
SGI-Challenge	200	23.9	20.9-27.7	-O2, mips2, -shared	29.8	25.0-36.2	**[15]	n.a.
IBM-RS/590	67	21.4	13.9-31.0	-O, pwr2	27.1	23.1-33.7	***[16]	27
IBM-RS/390	67	24.3	18.4-33.3	-O, pwr2	24.6	18.4-33.2	O3, pwr2	24

Appendix VII) GEXAM1 measurements

Table 10: GEXAM1 runs on CERNVM
using 10 or 100 events and RDNM or RANLUX code

No. Events	Total time	Sec/event	comments
10	20.75	1.7813	Original GRNDM
100	178.51	1.7601	Original GRNDM
10	19.64	1.6625	RANLUX
100	167.80	1.6525	RANLUX

[14] -O3, -mips2, -non_shared, -static
[15] -O3, -mips2, -non_shared
[16] -O3, -qarch=pwr2, -Q (Inlining allowed)

The small difference in the timings (between GRNDM and RANLUX) led us to believe that the number of calls to GRNDM in a GEXAM1 event is much smaller than for GALEPH. Checking the profile we could see that indeed this was the case: For 100 events GRNDM was called 4,981,902 times with 10,228,643 requests of random variables (i.e. 2 variables /call on the average), or about 50,000 calls per event. The comparable number was 2.75 millions per event for GALEPH.

As already mentioned in Chapter 6, GEXAM1 is not really representative of actual HEP simulation jobs since it uses a very simplified detector which is far from being realistic, producing very little random "walk". Another explanation of the small difference, was the fact that the GRNDM used in this version of GEXAM1 was already converted to REAL*8 code making it rather fast on most machines.

GEXAM1 runs on HP using the Code Instrumentation option.

This option allows the insertion of code into the program so that execution profile statistics can be collected. When one relinks the program with the +P option procedures that call each other frequently can be placed near each other in the executable. This "procedure repositioning" should result in fewer misses in the instruction cache. Page faults and TLB misses should also be reduced when the code runs, (see HP Fortran/9000 Programmer's Guide, E0892, page 6-45).

We recompiled all the source of GEXAM1 with +O2 -K plus the +P option. We added +I in loader flags. The results were as follows:

- Using GEXAM1 (original run) → 1.507 seconds/event.
- Using GEXAMP (the improved procedure order) → 1.422 seconds/event

In other words, this gave an overall improvement of about 5 % but no more .

Table 11: Summary run of GEXAM1 (315 and 321) [100 events]

	CERN	GEXAM1/315		GEXAM1/321	
Platforms	Units	Options	time	Options	time
IBM/ES9000	20	Opt(3)	1.76	----	----
IBM/600J	9	Opt(3)	3.94		
HP-712 (80 MHz)	21.0	+O2	2.32	+O2	2.86
HP-735 (99 MHz)	27.0	+O2, +I, +P	1.35		1.76
HP-735 (125 MHz)	34.0	+O2	1.09	+O2	1.39
HP-819 (100 MHz)	30.0	+O2, DA1.1d	1.31	+O3 DA1.1d	1.59
DEC-3400	18.0	-O2, static	3.18	-O2	4.63
DEC-3900	52.0	-O2	1.41	-O2	1.80
SGI-Challenge (150 MHz)	20.0	-O2, -mips2	2.93	-O2, -mips2	3.04
SGI-Challenge (200 MHz)	29.0	-O2, -mips2	2.29	-O2, -mips2	2.40[17]
IBM-RS/390	27.0	-O2	2.21	-O2	2.75
IBM-RS/390	27.0	-O3, -Q, -pwr2[18]	2.058		

[17] Using same module as SGI (150 MHz).

[18] Using the most powerful options (on CERNSP2) we only got 6-7 % improvement in execution time.

Appendix VIII) VZERO and unrolling

During the discussions of VZERO (See section 7.3) we were offered an optimised version of VZERO by B.Montgomery (HP/US). This version used double precision (REAL*8) store operations instead of REAL*4. We decided to compare this version (MZERO) to the orinigal VZERO from the libary on the various RISC platforms. Given our success with IUHUNT, we also decided to try to unroll the loops in both VZERO and MZERO. The results have been summarised in Appendix Tables 12, 13, and 14. The 4 runs represent 4 different array lengths, forcing various alignments: a(10004), b(10003), c(10001), d(10000).

Table 12. VZERO (original code) compared to MZERO
(10,000 iterations)

HP-735/99		DEC-3400		RS6000-590	
VZERO	MZERO	VZERO	MZERO	VZERO	MZERO
a) 2.04	1.01	4.17	2.47	3.02	1.50
b) 2.03	1.01	4.22	2.53	3.03	1.51
c) 2.02	1.02	4.34	2.53	3.03	1.51
d) 2.04	1.02	4.05	2.52	3.01	1.52

In Table 12 we see that MZERO was consistently twice as fast as VZERO (slightly less on DEC). We then unrolled VZERO, keeping MZERO unchanged.

Table 13. VZERO (Unrolled code) compared to MZERO.

HP-735/99		DEC-3400		RS6000-590	
VZERO	MZERO	VZERO	MZERO	VZERO	MZERO
a) 2.04	1.02	1.43	2.53	0.77	1.54
b) 2.03	1.02	1.48	2.53	0.77	1.53
c) 2.03	1.01	1.52	2.53	0.76	1.54
d) 2.03	1.01	1.60	2.48	0.77	1.52

In this case the unrolled VZERO beat MZERO on DEC and IBM, but not on HP. We then decided to unroll MZERO as well.

Table 14 VZERO (Unrolled 4x), compared to MZERO (Unrolled 2x)

HP-735/99		DEC-3400		RS6000-590	
VZERO	MZERO	VZERO	MZERO	VZERO	MZERO
a) 2.05	1.02	1.48	1.47	0.77	0.78
b) 2.03	1.02	1.50	1.47	0.77	0.77
c) 2.02	1.02	1.45	1.43	0.76	0.78
d) 2.02	1.01	1.55	1.43	0.77	0.78

Interestingly enough, both programs executed with the same speed on DEC and IBM, but the situation remained unchanged on HP. This may explain why HP, US gave us their version in the first place. They are aware of its importance.

The listing of both unrolled versions:

```
    SUBROUTINE MZERO (A, N)
 c This version uses real*8 stores, but assumes the initial vector
 c is either real*4 or integer        Bob Montgomery (HP/US DEC.
94)
     integer a(n)
     real*8 b(1)
     pointer (p, b)
 c
     if(n.le.0) return
     if(n.lt.1024)then
      do i =1, n
       a(i)=0
      end do
      return
     endif

     nl = n
     ia= loc(a(1))
     if(mod(ia, 8).eq.0) then
 c      ***a(1) is aligned on a 8-byte boundry
        p=ia
     else
 c     ***a(1) is aligned on a 4-byte boundry
        p=loc(a(2))
        a(1)=0
        nl=nl-1
     endif
 c ---prepare for real*8 clear--
       n8=nl/2
      it=nl-(n8*2)
 cry --try to unroll 2-fold---
      nn8=mod(n8, 2)
      do 17 I=1, nn8
      b(i)=0.0d0
 17    continue
      do 19 i=1+nn8, n8, 2
      b(i)=0.0d0
      b(i+1)=0.0d0
 19    continue

      if(it.gt.0)a(n)=0
      return
      end
```

```
         SUBROUTINE VZERO(A, N)
cry----An unrolled version ---
-
      INTEGER  A(*)
      IF (N.LE.0) RETURN
       ii=mod(n, 4)
       do 8 i=1, ii
       a(i)=0
 8    continue
        DO 9 I= 1+ii, N, 4
      A(I)= 0
      A(I+1)= 0
       A(I+2)= 0
      A(I+3)= 0
 9   continue
      return
       end
```

Appendix IX. Profiling of the full DICE code (10 events on HP-735/99)

Table 15. Profiling of DICE execution

Using Original GRNDM				RANLUX version of GRNDM			
%time	seconds	calls	name	%time	seconds	calls	name
14.7	423.99	24845520	gtnext_	15.9	527.60	28392177	gtnext_
4.9	140.99	101632183	gmepos_	5.7	188.11	106566655	gmepos_
4.5	130.24	64469055=>	grndm	4.1	135.88	73134962	grndm_
8.0	230.60	=>	$$divoI				
4.0	116.71	=>	$$mulI				
4.3	124.56	16796893	gtmedi	5.2	171.31	19401823	gtmedi_
3.4	97.72	134720604	FTN_DTAN	4.0	132.49	144167086	FTN_SIGN$
3.1	89.38	3175934	gmedia_	4.1	135.87	3678394	gmedia_
2.7	78.48	3175934	gtrack_	3.6	119.64	3678394	gtrack_
2.4	69.00		gustep_	2.6	87.81		gustep_
2.3	66.30	21280343	gtgama_	2.8	94.45	23897993	gtgama_
2.2	64.11	5342602	gschit_	2.7	90.97	5514518	gschit_
1.8	53.47		glandz_	2.2	73.12		glandz_
1.7	48.87		gusspy_	1.6	52.74		gusspy_
1.7	47.92		gtelec_	2.1	68.56		gtelec_
1.5	44.48		gnotub_	1.8	58.52		gnotub_
1.5	42.86		mzneed_	1.8	60.65		mzneed_
1.4	41.64		gnobox_	1.5	50.62		gnobox_
1.3	37.57		gmolie_	1.6	52.99		gmolie_
1.0	28.59		gncone_	1.4	47.05		gncone_
0.9	25.36		uhtoc_	1.0	33.55		uhtoc_
0.9	24.83		gtrmul_	1.0	33.01		gtrmul_

Observations:

The 3 lines marked with => belong to the same subroutine call as GRNDM since these are calls to millicode functions used almost exclusively by this subroutine This means that GRNDM was actually consuming 16.5% of the total time in the original run compared to 4.1% when the RANLUX version was invoked.

The total reduction in time per event was about 10% improvement (from 300 to 266 seconds).
