Compiler Project

José Miguel Dana Pérez
Supervisor: Sverre Jarp

September 21, 2005

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this project we have done a study of the most popular compilers of `C/C++` today (`gcc`[1] and `icc`[2]) using several tests extracted from `ROOT`[3], `Geant4`[4] and `CLHEP`[5].

The extracted battery of tests is described below:

- `ROOT`:
  - `TGeoArb8::Contains(...)`
  - `TGeoCone::Contains(...)`
  - `TRandom::Landau(...)`
  - `TRandom3::Rndm(...)`
- `Geant4`:
  - `G4AffineTransform::InverseProduct(...)`
  - `G4Mag::EvaluateRhsGivenB(...)`
  - `G4Tubs::Inside(...)`
- `CLHEP`:
  - `HepMatrix::invertHaywood5(...)`
  - `RanluxEngine::flat(...)`
  - `HepRotation::RotateX(...)/RotateY(...)/RotateZ(...)`

For every test we have taken times using Itanium 2 and Xeon platforms (detailed in Appendix A).

Chapter 2

The timing library

We have develop a `timing` library to measure the time necessary for every execution.

We have implemented functions that return the spent `Real`[1], `User`[2] and `System Time`[3].

Also, we have develop a function that returns the number of cycles spent by the machine, reading the `RDTSC` (in x86 architectures) and the `ITC` register (in Itanium architectures). The assembly code used by this function has been tested for Itanium, Xeon and Pentium IV architectures using `icc` and `gcc` compilers.

[1] The total time.
[2] The time dedicated to computational tasks.
[3] The time dedicated to I/O, context changes, etc.

Chapter 3

ROOT

3.1 TGeoArb8::Contains(...)

This function is a geometrical function. It takes the vertices of a polygon and the coordinates of a point and evaluates if this is inside or outside the polygon.

This is a computational function, all the work is done by the processor and its ALU.

We can see the results in the tables 3.1 and 3.2.

Table 3.1: ROOT::TGeoArb8 in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	23.913 (100.0%)	24.144 (100.9%)	19.103 (79.80%)	21.107 (88.20%)	8.9508 (37.40%)	14.051 (58.70%)
-O3	23.211 (100.0%)	23.195 (99.90%)	20.539 (88.40%)	9.0506 (38.90%)	14.117 (60.80%)	7.8834 (33.90%)
-O2 + -ipo					18.634 (100.0%)	64.472 (345.9%)
-O2 + -finline-functions	24.445 (100.0%)	22.794 (93.20%)	20.538 (84.00%)	20.325 (83.10%)	18.635 (76.20%)	14.062 (57.50%)

Table 3.2: ROOT::TGeoArb8 in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	5.62 (100.0%)	10.63 (189.1%)	11.77 (209.4%)	11.7 (208.1%)	5.38 (95.70%)	5.52 (98.20%)
-O3	5.69 (100.0%)	8.11 (142.5%)	11.78 (207.0%)	5.82 (102.2%)	5.39 (94.70%)	5.52 (97.00%)
-O2 + -ipo					5.21 (100.0%)	5.2 (99.80%)
-O2 + -finline-functions	5.62 (100.0%)	10.59 (188.4%)	11.78 (209.6%)	11.65 (207.2%)	5.22 (92.80%)	5.53 (98.30%)

We can see a strange time for `icc 9.0` and `-O2 + -ipo` compilation flags in table 3.1. In this case the algorithm requires more than three times the time required for `icc 8.1`.

3.2 TGeoCone::Contains(...)

This function is very similar to the previous one, a geometrical function that returns if a point is inside a cone or not.

In summary, high computational power and low memory access.

In the tables 3.3 and 3.4 is possible to show the obtained results:

Table 3.3: ROOT::TGeoCone in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	32.226 (100.0%)	31.224 (96.80%)	25.198 (78.10%)	27.051 (83.90%)	23.542 (73.00%)	23.195 (71.90%)
-O3	29.202 (100.0%)	31.224 (106.9%)	25.211 (86.30%)	27.05 (92.60%)	20.203 (69.10%)	19.203 (65.70%)
-O2 + -ipo					23.195 (100.0%)	22.193 (95.60%)
-O2 + -finline-functions	32.561 (100.0%)	31.892 (97.90%)	25.197 (77.30%)	26.699 (81.90%)	22.193 (68.10%)	22.194 (68.10%)

Table 3.4: ROOT::TGeoCone in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	9.21 (100.0%)	13.66 (148.3%)	13.59 (147.5%)	11.76 (127.6%)	10.16 (110.3%)	10.23 (111.0%)
-O3	9.38 (100.0%)	11.99 (127.8%)	13.92 (148.4%)	11.76 (125.3%)	10.15 (108.2%)	10.23 (109.0%)
-O2 + -ipo					10.15 (100.0%)	10.22 (100.6%)
-O2 + -finline-functions	9.28 (100.0%)	13.66 (147.1%)	13.57 (146.2%)	11.77 (126.8%)	10.14 (109.2%)	10.28 (110.7%)

3.3 TRandom::Landau(...)

The `TRandom` class contains a lot of functions to generate random numbers. In this case, this is not a real random number generator, it generates a random number following a Landau distribution with mpv(most probable value) and sigma.

In this case, the problem has a bottleneck due to the elevate memory use (we have a huge table used by the `Landau` function and it may be statically allocated in memory).

The results are in the tables 3.5 and 3.6.

Table 3.5: ROOT::TRandom in Itanium 2 architecture

	gcc 3.2.3	**gcc 3.4.4**	**gcc 4.0.1**	**gcc 4.1.0**	**icc 8.1**	**icc 9.0**
-O2	22.525 (100.0%)	22.632 (100.4%)	22.565 (100.1%)	35.115 (155.8%)	28.707 (127.4%)	22.369 (99.30%)
-O3	22.421 (100.0%)	22.605 (100.8%)	22.531 (100.4%)	22.59 (100.7%)	31.85 (142.0%)	22.364 (99.70%)
-O2 + -ipo					28.701 (100.0%)	22.368 (77.90%)
-O2 + -finline-functions	22.55 (100.0%)	22.632 (100.3%)	22.532 (99.90%)	22.59 (100.1%)	28.691 (127.2%)	22.354 (99.10%)

Table 3.6: ROOT::TRandom in Xeon architecture

	gcc 3.2.3	**gcc 3.4.4**	**gcc 4.0.1**	**gcc 4.1.0**	**icc 8.1**	**icc 9.0**
-O2	35.28 (100.0%)	46.05 (130.5%)	46.92 (132.9%)	45.13 (127.9%)	34.9 (98.90%)	34.32 (97.20%)
-O3	35.18 (100.0%)	45.12 (128.2%)	45.86 (130.3%)	46.3 (131.6%)	34.22 (97.20%)	35.64 (101.3%)
-O2 + -ipo					34.49 (100.0%)	34.65 (100.4%)
-O2 + -finline-functions	35.79 (100.0%)	45.13 (126.0%)	45.18 (126.2%)	45.78 (127.9%)	34.53 (96.40%)	33.47 (93.50%)

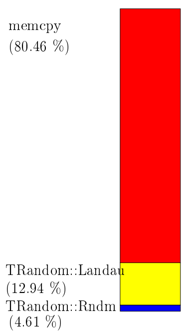

Figure 3.1: Problem with memory access.

In this algorithm is really important improve the memory management, in the Figure 3.1 we can see how more than the 80% of the total time is necessary due the `memcpy` function. The **real** algorithm is only a 13% of the total time[1].

[1] For the `gcc 3.2.3` compiler

3.4 TRandom3::Rndm(...)

This is a **real** number generator that uses the Mersenne Twistor method.

In this case, the algorithm doesn't use a huge table, is only computational power and we haven't the memory problem viewed in the previous analyzed function.

Table 3.7: ROOT::TRandom3 in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	16.914 (100.0%)	17.294 (102.2%)	17.592 (104.0%)	17.248 (101.9%)	12.927 (76.40%)	13.811 (81.60%)
-O3	16.923 (100.0%)	17.296 (102.2%)	17.592 (103.9%)	17.264 (102.0%)	13.138 (77.60%)	13.381 (79.00%)
-O2 + -ipo					13.251 (100.0%)	13.706 (103.4%)
-O2 + -finline-functions	16.926 (100.0%)	17.282 (102.1%)	17.592 (103.9%)	17.247 (101.8%)	12.918 (76.30%)	13.794 (81.40%)

Table 3.8: ROOT::TRandom3 in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	7.05 (100.0%)	6.93 (98.20%)	7.07 (100.2%)	7.16 (101.5%)	6.61 (93.70%)	7.46 (105.8%)
-O3	7.07 (100.0%)	6.92 (97.80%)	7.07 (100.0%)	7.17 (101.4%)	7.47 (105.6%)	7.46 (105.5%)
-O2 + -ipo					6.62 (100.0%)	7.49 (113.1%)
-O2 + -finline-functions	7.05 (100.0%)	6.93 (98.20%)	7.06 (100.1%)	7.14 (101.2%)	6.62 (93.90%)	7.45 (105.6%)

Chapter 4

GEANT4

4.1 G4AffineTransform::InverseProduct(. . .)

This function implements the inverse product of two matrix and store the result into another one.

In this problem we have a lot of floating point multiplications, but the matrices are really small and we can work with the cache memory all the time.

Table 4.1: GEANT4::G4AffineTransform in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	11.154 (100.0%)	11.154 (100.0%)	10.888 (97.60%)	10.887 (97.60%)	14.56 (130.5%)	14.551 (130.4%)
-O3	11.155 (100.0%)	11.154 (99.90%)	10.88 (97.50%)	10.887 (97.50%)	14.551 (130.4%)	14.555 (130.4%)
-O2 + -ipo					14.56 (100.0%)	14.56 (100.0%)
-O2 + -finline-functions	11.155 (100.0%)	11.153 (99.90%)	10.88 (97.50%)	10.887 (97.50%)	14.559 (130.5%)	14.561 (130.5%)

Table 4.2: GEANT4::G4AffineTransform in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	3.05 (100.0%)	7.31 (239.6%)	12.49 (409.5%)	10.15 (332.7%)	3.71 (121.6%)	3.65 (119.6%)
-O3	3.12 (100.0%)	5.69 (182.3%)	12.48 (400.0%)	10.15 (325.3%)	3.71 (118.9%)	3.66 (117.3%)
-O2 + -ipo					3.71 (100.0%)	3.65 (98.30%)
-O2 + -finline-functions	3.05 (100.0%)	7.3 (239.3%)	12.49 (409.5%)	10.15 (332.7%)	3.71 (121.6%)	3.65 (119.6%)

4.2 G4Mag::EvaluateRhsGivenB(. . .)

This function returns the value of the magnetic field B and calculates the value of the derivative dydx.

Table 4.3: GEANT4::G4Mag in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	29.555 (100.0%)	28.854 (97.60%)	23.009 (77.80%)	23.044 (77.90%)	9.4198 (31.80%)	8.8128 (29.80%)
-O3	23.945 (100.0%)	23.676 (98.80%)	22.843 (95.30%)	17.723 (74.00%)	8.4178 (35.10%)	7.8164 (32.60%)
-O2 + -ipo					8.3829 (100.0%)	8.1782 (97.50%)
-O2 + -finline-functions	24.364 (100.0%)	24.063 (98.70%)	22.844 (93.70%)	17.733 (72.70%)	8.3826 (34.40%)	8.8126 (36.10%)

Table 4.4: GEANT4::G4Mag in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	8.96 (100.0%)	12.74 (142.1%)	12.43 (138.7%)	12.7 (141.7%)	9.41 (105.0%)	9.08 (101.3%)
-O3	8.18 (100.0%)	9.21 (112.5%)	12.42 (151.8%)	12.39 (151.4%)	9.4 (114.9%)	9.09 (111.1%)
-O2 + -ipo					9.1 (100.0%)	9.09 (99.80%)
-O2 + -finline-functions	8.81 (100.0%)	12.2 (138.4%)	12.41 (140.8%)	12.42 (140.9%)	9.09 (103.1%)	9.09 (103.1%)

4.3 G4Tubs::Inside(...)

We have, again, a geometric function that return if a vector is inside, outside or in the surface of a tube.

In the first implementation of test we didn't use the return of the function at the end of the `main`, the result of the execution is in the Table 4.5.

Table 4.5: GEANT4::G4Tubs in Itanium 2 architecture (if we don't use the return)

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	25.713 (100.0%)	26.701 (103.8%)	26.716 (103.9%)	27.049 (105.1%)	26.049 (101.3%)	0.44509 (1.700%)
-O3	24.378 (100.0%)	27.048 (110.9%)	26.699 (109.5%)	27.7 (113.6%)	26.032 (106.7%)	0.4452 (1.800%)
-O2 + -ipo					0.44503 (100.0%)	0.44554 (100.1%)
-O2 + -finline-functions	25.031 (100.0%)	26.716 (106.7%)	26.715 (106.7%)	27.716 (110.7%)	0.44527 (1.700%)	0.44531 (1.700%)

As we can see, the `icc 9.0` compiler (and `icc 8.1` with `-ipo` or `-finline-functions` flag) discovers that the return of the function is not necessary and doesn't compute this, obtaining a execution time of 1.7% respect the time taken by `gcc 3.2.3`.

In Tables 4.6 and 4.7 we have the results using the return of the function an forcing to `icc` compilers to process it. In this case, `gcc` compilers have the best improve (for Itanium 2 architecture).

Table 4.6: GEANT4::G4Tubs in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	20.358 (100.0%)	22.375 (109.9%)	24.043 (118.1%)	23.877 (117.2%)	26.548 (130.4%)	27.534 (135.2%)
-O3	19.869 (100.0%)	22.194 (111.7%)	24.044 (121.0%)	23.878 (120.1%)	26.533 (133.5%)	26.55 (133.6%)
-O2 + -ipo					26.548 (100.0%)	26.55 (100.0%)
-O2 + -finline-functions	20.37 (100.0%)	22.361 (109.7%)	23.71 (116.3%)	23.863 (117.1%)	26.549 (130.3%)	27.552 (135.2%)

Table 4.7: GEANT4::G4Tubs in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	6.71 (100.0%)	8.56 (127.5%)	9.03 (134.5%)	8.16 (121.6%)	7.34 (109.3%)	4.82 (71.80%)
-O3	6.86 (100.0%)	9.07 (132.2%)	9.03 (131.6%)	8.19 (119.3%)	7.34 (106.9%)	4.83 (70.40%)
-O2 + -ipo					7.35 (100.0%)	4.82 (65.50%)
-O2 + -finline-functions	6.68 (100.0%)	8.51 (127.3%)	9.05 (135.4%)	8.18 (122.4%)	7.36 (110.1%)	4.83 (72.30%)

We obtain a really good time for `icc 9.0` and Xeon architecture (see table 4.7).

Chapter 5

CLHEP

5.1 HepMatrix::invertHaywood5(...)

This function get an input matrix of 5x5 and calculates its inverse, returning it.

This operation uses a lot of local variables and a big vector to generate the output matrix.

Table 5.1: HepMatrix::invertHaywood5 in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	3.8455 (100.0%)	3.5846 (93.20%)	2.9668 (77.10%)	2.8597 (74.30%)	32.793 (852.7%)	2.9509 (76.70%)
-O3	3.6318 (100.0%)	3.6986 (101.8%)	2.9665 (81.60%)	2.8613 (78.70%)	32.581 (897.1%)	2.8469 (78.30%)
-O2 + -ipo					35.838 (100.0%)	35.479 (98.90%)
-O2 + -finline-functions	3.9008 (100.0%)	3.5808 (91.70%)	2.9659 (76.00%)	2.86 (73.30%)	35.962 (921.9%)	2.9508 (75.60%)

Table 5.2: HepMatrix::invertHaywood5 in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	5.22 (100.0%)	16.21 (310.5%)	18.83 (360.7%)	20.67 (395.9%)	4.37 (83.70%)	4.44 (85.00%)
-O3	4.84 (100.0%)	9.13 (188.6%)	18.89 (390.2%)	20.62 (426.0%)	4.4 (90.90%)	4.43 (91.50%)
-O2 + -ipo					23.18 (100.0%)	22.1 (95.30%)
-O2 + -finline-functions	5.25 (100.0%)	16.22 (308.9%)	18.85 (359.0%)	20.68 (393.9%)	4.41 (84.00%)	4.46 (84.90%)

I think that this function is a interesting function to study (maybe the most interesting function in all the report).

In table 5.1 we can see a really bad time for `icc 8.1`, however, for Xeon architecture it is the best one. Other interesting thing is that `icc` compilers obtain really bad results with `-ipo` flag in both architectures.

5.2 RanluxEngine::flat(. . .)

This function returns a pseudo random number in the open interval (0,1). We are, again, in front of a computational task.

Table 5.3: RanluxEngine::flat in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	55.393 (100.0%)	54.298 (98.00%)	51.948 (93.70%)	51.982 (93.80%)	23.13 (41.70%)	17.413 (31.40%)
-O3	54.042 (100.0%)	46.981 (86.90%)	51.843 (95.90%)	51.841 (95.90%)	19.781 (36.60%)	13.779 (25.40%)
-O2 + -ipo					22.303 (100.0%)	22.305 (100.0%)
-O2 + -finline-functions	55.392 (100.0%)	52.723 (95.10%)	51.843 (93.50%)	51.812 (93.50%)	22.42 (40.40%)	17.414 (31.40%)

Table 5.4: RanluxEngine::flat in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	30.28 (100.0%)	36.87 (121.7%)	28.34 (93.50%)	27.93 (92.20%)	38.37 (126.7%)	33.53 (110.7%)
-O3	30.17 (100.0%)	36.07 (119.5%)	28.45 (94.20%)	27.06 (89.60%)	38.39 (127.2%)	34.26 (113.5%)
-O2 + -ipo					37.5 (100.0%)	33.72 (89.90%)
-O2 + -finline-functions	30.09 (100.0%)	35.87 (119.2%)	28.03 (93.10%)	27.05 (89.80%)	37.68 (125.2%)	34 (112.9%)

The results for `icc` compilers are really good for this algorithm and Itanium architecture.

5.3 HepRotation::RotateX(...)/RotateY(...)/RotateZ(...)

This function rotates a `HepRotation` object using simple floating point operations.

In the Tables 5.5 and 5.6 we can see the results of the execution.

Table 5.5: HepRotation::Rotate in Itanium 2 architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	32.396 (100.0%)	32.526 (100.4%)	32.527 (100.4%)	32.66 (100.8%)	14.018 (43.20%)	13.826 (42.60%)
-O3	30.59 (100.0%)	29.986 (98.00%)	29.722 (97.10%)	29.636 (96.80%)	14.018 (45.80%)	13.817 (45.10%)
-O2 + -ipo					2.338 (100.0%)	2.0025 (85.60%)
-O2 + -finline-functions	30.704 (100.0%)	29.853 (97.20%)	29.722 (96.80%)	29.654 (96.50%)	2.136 (6.900%)	13.827 (45.00%)

Table 5.6: HepRotation::Rotate in Xeon architecture

	gcc 3.2.3	gcc 3.4.4	gcc 4.0.1	gcc 4.1.0	icc 8.1	icc 9.0
-O2	65.39 (100.0%)	73.87 (112.9%)	73.88 (112.9%)	74.65 (114.1%)	49.75 (76.00%)	20.34 (31.10%)
-O3	64.82 (100.0%)	73.72 (113.7%)	77.13 (118.9%)	77.24 (119.1%)	49.81 (76.80%)	20.38 (31.40%)
-O2 + -ipo					17.75 (100.0%)	20.36 (114.7%)
-O2 + -finline-functions	64.76 (100.0%)	76.97 (118.8%)	77.14 (119.1%)	77.24 (119.2%)	17.68 (27.30%)	20.4 (31.50%)

Again, we obtain very good times for `icc` compilers.

Chapter 6

Conclusions

In summary, we can say that the `icc` compiler is a bit unstable (in fact, it is under development). It is really good with some tasks, specially where the memory access is the bottleneck. Also, we obtain really good results with some geometric functions. However, when we have tasks with high computational requisites (like random number generators) `icc` can take a lot of time, if we compare with `gcc`.

We should try with different compilation flags, take a look of the generated assembly code and study the internal architecture (and, maybe, the code) of the respective compilers to give better conclusions. In fact, we can't be sure about the *reasons* that convert one compiler in the best one, we only know the *situations* in which one compiler is better than others.

Appendix A

Architectures

We have used the architectures described below for all the tests (the specification is for every node of the respective cluster).

- **oplaslim1**:
  - *Linux Distribution*: Scientific Linux CERN release 3.0.5.
  - *Linux Version*: 2.4.21-32.0.1.EL.cern
  - *CPU*: Intel Xeon 64 bits 3.60 GHz (two per node).
  - *Cache Memory*: 2048 KB (per CPU).
  - *Main Memory*: 7 GB.
- **oplapro21**:
  - *Linux Distribution*: Scientific Linux CERN release 3.0.5.
  - *Linux Version*: 2.6.12.2
  - *CPU*: Intel Itanium 2 64 bits 1.50 GHz (two per node).
  - *System Bus Bandwidth*: 6.4 GB/s.
  - *Cache Memory*:
    - *L1*: 32KB.
    - *L2*: 256 KB.
    - *L3*: 6 MB.
  - *Main Memory*: 2 GB.
  - *Main Memory Bus Bandwidth*: 6.4 GB/s.

Appendix B

Compilers

We have used these versions of the `gcc` and `icc` compilers:

- `gcc 3.2.3`
- `gcc 3.4.4`
- `gcc 4.0.1`
- `gcc 4.1.0`
- `icc 8.1`
- `icc 9.0`

Appendix C

Optimization Flags

Every version of `gcc` or `icc` has their own group of flags for every level of optimization. In this Appendix, we use like example the optimization of flags of `gcc 4.0.1`, described below:

- -O2:
  - `-fdefer-pop` (from -O1)
  - `-fdelayed-branch` (from -O1)
  - `-fguess-branch-probability` (from -O1)
  - `-fcprop-registers` (from -O1)
  - `-floop-optimize` (from -O1)
  - `-fif-conversion` (from -O1)
  - `-fif-conversion2` (from -O1)
  - `-ftree-ccp` (from -O1)
  - `-ftree-dce` (from -O1)
  - `-ftree-dominator-opts` (from -O1)
  - `-ftree-dse` (from -O1)
  - `-ftree-ter` (from -O1)
  - `-ftree-lrs` (from -O1)
  - `-ftree-sra` (from -O1)
  - `-ftree-copyrename` (from -O1)
  - `-ftree-fre` (from -O1)
  - `-ftree-ch` (from -O1)

  - -fmerge-constants (from -O1)
  - -fthread-jumps
  - -fcrossjumping
  - -foptimize-sibling-calls
  - -fcse-follow-jumps
  - -fcse-skip-blocks
  - -fgcse
  - -fgcse-lm
  - -fexpensive-optimizations
  - -fstrength-reduce
  - -frerun-cse-after-loop
  - -frerun-loop-opt
  - -fcaller-saves
  - -fforce-mem
  - -fpeephole2
  - -fschedule-insns
  - -fschedule-insns2
  - -fsched-interblock
  - -fsched-spec
  - -fregmove
  - -fstrict-aliasing
  - -fdelete-null-pointer-checks
  - -freorder-blocks
  - -freorder-functions
  - -funit-at-a-time
  - -falign-functions
  - -falign-jumps
  - -falign-loops
  - -falign-labels
  - -ftree-pre

- -O3:

  - `-O2`
  - `-finline-functions`
  - `-funswitch-loops`
  - `-fgcse-after-reload`

- `-O2 + -ipo` (only icc):
  - `-O2`
  - `-ipo`

- `-O2 + -finline-functions`:
  - `-O2`
  - `-finline-functions`

For more information, you can visit the documentation section into the webpage of the `gcc`[1] project.

Bibliography

[1] *GCC compiler*, http://gcc.gnu.org

[2] *Intel icc compiler*, http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/index.htm

[3] *ROOT, An Object Oriented Data Analysis Framework*, http://root.cern.ch

[4] *Geant4*, http://cern.ch/geant4/

[5] *CLHEP, A Class Library for High Energy Physics*, http://wwwasd.web.cern.ch/wwwasd/lhc++/clhep/