

Searching for optimal performance on IPF/Linux

Sverre Jarp, HP Labs
27 August 2002

1.0 Introduction

The Itanium Processor Family is based on the EPIC (Explicitly Parallel Instruction Computing) architecture, which comes with a rich set of architectural enhancements for improving performance. In particular, the architecture dictates that instruction scheduling be done explicitly via 3-slot bundles. The “micro-architecture” decides how many bundles can be issued in parallel. Both Itanium and Itanium 2 can issue two in parallel, so that the peak speed becomes six instructions per cycle (IPC).

The difficult task of filling the bundles with useful work (and not just NOP instructions) falls on the compilers. They must find sufficient parallelism (in a sustained manner), they must sort out all inter-dependencies between instructions, and finally they must perform detailed instruction scheduling with an in-depth knowledge of the micro-architecture of the target IPF processor. In particular, they must hide cache/memory latencies to the best of their ability.

If RISC was regarded as “Relegate Important Stuff to Compilers”, EPIC can be said to be (Relegate Important Stuff to Compilers)²! The implications, for application programmers, are that the use of advanced compiler options is even more important than before in order to achieve optimal execution speed.

This paper looks at the Intel and GNU compilers and their respective options for obtaining peak performance on IPF. It also stresses the importance of using profiling tools, such as gprof, for identifying bottlenecks and unexpected hot spots. Finally, it discusses certain review and intervention techniques for cases where the compilers do not do optimize automatically.

The paper does not discuss VTUNE, which is Intel’s performance monitoring tool (a commercial product), or PFMON, which is a performance-monitoring tool for IPF/Linux. The latter is reviewed in detail in my paper “A Methodology for using the Itanium 2 Performance Counters for Bottleneck Analysis”.

2.0 The Intel compilers (Release 6) and their performance options

2.1 ecc/efc performance-related options

Here is a list of the most relevant performance-oriented compiler options for the Intel/Linux compilers. For further information, please see the Intel C/C++ Compiler User’s Guide (CL-600-06) and Intel FORTRAN Compiler User’s Guide (FL-600-06).

Option	Explanation
<i>O2 (O)</i>	Standard Optimization
<i>O3</i>	O plus High Level Optimization
<i>ftz</i>	Flush (denormalized numbers) to zero
<i>unroll0</i>	Disable unrolling
<i>ipo</i>	Multi-file inter-procedural optimization
<i>prof_gen, prof_use</i>	Profile-guided optimization (NB: needs two passes)
<i>fno-alias</i>	Assume the pointers are not aliased
<i>fno-fnalias</i>	Assume pointers are not aliased inside functions

2.2 ecc/efc options in practice

In order to understand better the impact of a given performance option, we try them out in a test. The selected test case is *eon*, the only defender of C++ in the current SPECint suite. In formal SPEC testing, *eon* is run with three different rendering algorithms, Kajiya, Cook, and Rushmeier. For reason of simplicity, I have only used the first one, invoking the program via the command:

```
eon chair.control.kajiya chair.camera chair.surfaces out1 ppm out3
```

The following table summarizes the relative results for *eon* on an HP zx2600 system (with a 16 KB page size) using the ecc 6.0 compiler:

	Relative speed
<i>ecc -O2</i>	1.00
<i>ecc -O3</i>	1.00
<i>ecc -O3 -ftz</i>	1.00
<i>ecc -O3 -ftz -unroll0</i>	0.93
<i>ecc -O3 -ftz -prof_use</i>	1.02
<i>ecc -O3 -ftz -ipo</i>	1.37
<i>ecc -O3 -ftz -prof_use -ipo</i>	1.54

2.3 ecc/efc options: the bottom line

O2/O3: Normally one would start with the former, and then move to the latter and compare. O3 does not always give a better result as seen in this example (and also documented in Intel's own help message). This is mainly because the High Level Optimizer adds advanced features, such as loop transformations and prefetching, which may do more harm than good in certain contexts.

It is also worth noting the O3/O2 ratio may change when combined with other options, such as *ipo* or *prof_use*. This is the reason why the tests were continued with O3 and not O2.

ftz: It is recommended to use this option by default for programs which contain floating-point (FP) calculations. It tells the compiler to flush denormalized floating-point numbers to zero. In certain cases where such numbers are too small to be represented as a normal floating-point number, it may still be represented as something greater than zero. In such cases, IPF processors will escape to an expensive assist mode whenever the denormalized value is used in a calculation. In 99.9% of all cases, the users would be better off simply insisting that such a number be flushed to zero, rather than kept alive as a minute, denormalized quantity. In our case we see a very small improvement, but on a more floating-point intensive application, the difference might have been more significant.

unroll0: The Itanium version of the Intel compilers will unroll loops by default. In the test case, this option caused the performance to decrease by 7%, so I would definitely not recommend using this option by default. There may be other cases where the performance actually increases, however, so when hunting around for the last few percentage points, this option may do the trick.

ipo: This option tells the compiler to optimize beyond the scope of an individual procedure. It can do this, either as a "single file optimization" or a "multi-file optimization". The optimal use comes when the compiler optimizes across ALL input files in "multi-file" mode. For that to happen it is necessary to apply the option to all phases involved in creating the executable (for instance: compilation of FORTRAN source files, C source files, and the linkage phase). The compiler will then create a file containing the intermediate representation of the code in each compilation stage, so that during the linkage phase everything can be read in and given back to the compiler

for a global optimization. It must be noted that the compiler will not perform “multi-file optimization” if the build is done via libraries.

Performance increased by 37% compared to our initial run, so this is a very important option to use in all IPF compilations.

prof_gen/prof_use: These options are used for generating an execution profile in an initial run in order to optimize better during a second compilation. The process is called profile-guided optimization (PGO) and a simplified usage sequence might look like this:

```
ecc -O3 -prof_gen -o myprog myprog.c
./myprog < training.data           ← test run
ecc -O3 -prof_use -o myprog myprog.c
time ./myprog < production.data    ← production run
```

There are two possible drawbacks to this approach:

- The application must be compiled twice, so long compilations quickly become a drawback
- The “training data” (meant to represent the input data which steer the execution of the program) should exercise the same hot paths in the program as the “production data”

As a result, most people will tend to restrict the use of these options to the generation of “production executables”, i.e. versions, which will be around for a long time.

Using PGO (without IPO) gives a relatively disappointing result, since our example only improved by 2%. Combining it with ipo, however, we get the best result. The speed is now 54% above the base run.

fno-alias/fno-fnalias: These options need to be used with great care, because they inform the compiler that it can carry out aggressive optimizations based on the assumption that there will be no problems with pointer-aliasing, either inside the entire source file being compiled or inside functions (assuming aliasing across calls). The reason for mentioning the options is that, in certain cases, they may serve to re-optimize the innermost loop(s) of the program to get the best possible performance (provided the non-aliasing assumption holds).

This option was not tried in practice on eon.

Here are my conclusions in table format:

Intel compilers	Flags
Preferred	-O2/-O3 -ipo -prof_gen/-prof_use -ftz
Worth trying	-unroll0 -fno-alias/-fno-fnalias

3.0 The gcc compiler (Release 3.1) and its performance options

3.1 Important gcc performance-related options

As for ecc, I have established a list of some of the most relevant performance options. For further information, please see the gcc online documentation (<http://gcc.gnu.org/onlinedocs>).

Option	Explanation
<i>O1 (O)</i>	Default, non-aggressive optimization
<i>O2</i>	<i>O1</i> plus most other optimization techniques (but not function inlining and loop unrolling)
<i>O3</i>	<i>O2</i> plus function inlining (and register renaming)
<i>inline-functions</i>	Inline small functions (inside bigger ones)
<i>funroll-all-loops</i>	Perform unrolling of all loops
<i>ffast-math</i>	Allow optimizations that violate strict ANSI rules
<i>fexpensive-optimizations</i>	Perform minor optimizations which are relatively expensive
<i>fprefetch-loop-arrays</i>	Prefetch arrays inside loops

3.2 gcc options in practice

The following table summarizes the results for *eon* when using the g++ 3.1 compiler:

	Relative speed
<i>g++ -O1</i>	1.00
<i>g++ -O2</i>	1.23
<i>g++ -O3</i>	1.23
<i>g++ -O3 -funroll-all-loops</i>	1.30
<i>g++ -O3 -fprefetch-loop-arrays</i>	1.22
<i>g++ -O3 -ffast-math</i>	1.24
<i>g++ -O3 -fexpensive-optimizations</i>	1.24

3.3 gcc 3.1 options: the bottom line

O1/O2/O3: Unlike ecc, gcc defines the default optimization as *O1*. This is a conservative level where, for instance, the machine code sequence still respects the sequence of each high-level statement. In our tests we get significantly better results with *O2* and *O3* compared to *O1*, namely 23%. The difference between *O2* and *O3* is tiny, however. I would nevertheless recommend trying *O3*, but it is not guaranteed to improve the results on all applications.

inline-functions: This option tells the compiler to integrate all simple functions into their callers. This option is automatically included with *O3*, so there is normally no need to add this option by itself unless *O3* cannot be used for other reasons.

funroll-all-loops: This option unrolls all loops unconditionally. A “gentler” option, *funroll-loops*, will only unroll loops whose number of iterations can be determined at compile time or run time. In our test, we improved performance by a few additional percentage points by getting the compiler to unroll loops. This is not so surprising since we saw with the Intel compiler that *unroll0* (i.e. no unrolling) slowed down the performance. I should also mention that, with this particular test job, *funroll-loops* gave the same result as *funroll-all-loops*.

ffast-math: The most important optimization performed with this option, as far as IPF is concerned, is to “flush to zero” tiny floating-point numbers. I discussed its usefulness under the *ftz* option of the Intel compiler, so there is no need to repeat the rationale for using it. The option

does not impact our test job significantly, but I would nevertheless recommend including it with all applications that use floating-point calculations.

fprefetch-optimizations: This option is supposed to do a set of “expensive” optimizations which may increase compile time, and speed up execution time. Unfortunately, there seems to be no detailed documentation of what these optimizations are. I have never seen the option having a great impact on performance, so I remain skeptical as to its usefulness across a large set of applications.

fprefetch-loop-arrays: In principle, this option should have a positive effect because it enables prefetching of arrays inside loops, This can be of paramount importance for applications using vectors and matrices in a heavy way. Unfortunately, performance degraded slightly in our test case. I would, nevertheless, encourage people to try this option for all applications that feature loops which manipulate vectors or matrices. In chapter 5 we discuss a way of introducing manual prefetching.

Here are my conclusions in table format:

GNU compiler	Flags
Preferred	-O2/-O3 -ffast-math
Worth trying:	-funroll-all-loops/-funroll-loops -fprefetch-loop-arrays

4.0 Using the Linux profiling tool

4.1 Identifying the highest CPU-consuming routines

The Linux profiling tool, *gprof*, which is an old tool with several shortcomings, can nevertheless be used with little overhead to identify the most CPU-intensive routines in an executable. The following two tables, based on two runs of *eon* with two different sets of options, list all the routines that accumulate just over 50% of total CPU consumption:

ecc6 -O3 -ftz -p	Percent
GgGridIterator mrSurface Next	7.7
ggGridIterator<mrSurface*>::ComputeCellAndTraversal()	6.4
ggGridIterator<mrSurface*>::ComputeDistanceToPlanes()	5.8
ggRayXZRectangleIntersect()	5.6
mrSurfaceList::viewingHit()	4.9
ggPoint::ggPoint()	4.6
mrCookPixelRenderer::directLight()	3.9
ggSpectrum::operator=()	3.5
ggSpectrum::Set(float)	3.2
mrSurfaceList::shadowHit()	3.1
mrKajiyaPixelRenderer::kajiyaRadiance()	2.9
SUM	51.6

In this initial profile, created without *ipo* or *prof_use*, we observe that a lot of small routines, such as iterators, constructors and overloaded operators, consume a lot of time. It is particularly important in C++ programs to check if the report is dominated by such small routines, since they can often be inlined for better overall performance.

In this second table, however, we see that the *ipo* option has ensured that all the small routines have successfully been inlined and the remaining routines are listed as heavier consumers:

<code>ecc6 -O3 -ftz -p -ipo</code>	Percent
<code>mrSurfaceList::viewingHit()</code>	14.2
<code>mrKajiyaPixelRenderer::kajiyaRadiance()</code>	9.2
<code>ggRayXZRectangleIntersect()</code>	7.7
<code>mrGrid::viewingHit()</code>	7.6
<code>mrGrid::shadowHit()</code>	7.6
<code>mrCookPixelRenderer::directLight()</code>	7.5
SUM	53.8

The advantages of such inlining on an Itanium processor are twofold:

- By inlining the smaller routines the compiler is able to extract more ILP from the remaining (bigger) routines
- The branching to little routines (and back) is eliminated (as well as the code needed to initialize the environment inside the routine)

4.2 Identifying abnormal consumption

gprof can also be used across platforms to check that performance is “reasonable”. By, for instance, running the application under Linux/x86 and Linux/IPF a cross comparison of the profiles could indicate routines that consume a disproportionate amount of CPU time in one environment but not in the other. Such cases may not always be illegitimate, but they ought to trigger a close scrutiny of the source code and the corresponding machine code generated by the compiler.

5.0 Further search for IPF performance

Although IPF compilers have been in development for several years, it is no secret that they do not always optimize performance to full perfection. In such cases, it may be necessary to perform manual code inspection, paying attention to the following areas of optimization, such as:

- Loop optimization
- Data prefetching
- Disambiguation of pointers

5.1 Loop optimization

There are several features in the IPF architecture that assist in optimizing loops: predication, rotating registers, and branch instructions for control of software pipelining. Consequently it is vital to check that the compiler pipelines CPU-intensive loops. Such loops are easily identified by the combination of predicate usage and loop-controlling branch instruction as shown in the following example:

Original C-code:

```
for (i = i0; i < N; i+=2 )
  { if (min1>array[i]) min1=array[i]; if (min2>array[i+1]) min2=array[i+1]; }
```

Machine code generated by ecc version 6:

```

.bl_9:
{
  .mii
  (p16)    ld8    r32=[r10],16
  (p17)    cmp.gt.unc    p18,p0=r9,r33
  (p17)    cmp.gt.unc    p19,p0=r8,r35
} { .mmi
  (p16)    ld8    r34=[r3],16 ;;
  (p16)    lfetch [r2],16
  (p18)    mov    r9=r33
} { .mib    nop.m 0
  (p19)    mov    r8=r35
          br.ctop.sptk .bl_9 ;;

```

We recognize that this is a pipelined loop from the fact that every instruction (except *nop*'s and the *br.ctop*) have (rotating) predicates in front of them and that the loop ends with a *br.ctop* or *br.wtop*.

If the loops are not being pipelined, it may be necessary to modify the loops so that pipelining can take place. Normally, "modifying" means "simplifying" the loops. This can involve steps like:

- Removing if-constructs
- Disambiguating pointers
- Making sure the loop controlling index increases (rather than decreases)

Here is an example of a loop in need of pointer disambiguation. A year ago the following code was reported as not "pipelinable"¹:

```
for ( i = 1; i < *n; i++, k++ ) x[i] -=temp * ap[k] ;
```

The proposed improvement was the removal of **n* in the control of the loop iterations:

```
num = *n; for ( i = 1; i < num; i++, k++ ) x[i] -=temp * ap[k] ;
```

Although the current IPF compiler is able to pipeline the original loop (as a while loop) it is inefficient because the compiler has to assume that **n* can be modified inside the loop, so it gets reloaded in each iteration.

We also need to be aware of the fact that the IPF architecture can only pipeline the innermost loop when loops are nested. If this is not the right loop to be pipelined for attaining peak performance we may have to consider:

- Loop inversion (Exchanging the innermost loop with the next)
- Loop unrolling (Collapsing, for instance, the two innermost loops into one)

5.2 Prefetching

Both the Intel compiler at O3 and the GNU compiler with *fprefetch-loop-arrays* can perform prefetching. In many cases, however, this may turn out to be a sub-optimal strategy for a couple of reasons:

- Prefetch distance is wrong (too short, for instance)
- The data is prefetched to the wrong level in the cache hierarchy
- Non-trivial strides through the data

There may be cases where it is advisable to insert one's own prefetch instructions. The Intel compiler allows this to happen via intrinsics:

```
__lfetch(__MM_HINT_NT1, addr) ;
```

¹ Example shown in a presentation by K.Mazurkiewicz /Intel "ASC Itanium Performance Tuning Case Study" at IDF Fall 2001.

gcc allows prefetching via in-line assembly:

```
__asm__ volatile ("lfetch.nt1 [%0]" :: "r"(addr) : "memory");
```

5.3 Pointer disambiguation

When the compiler does not know if pointers point to overlapping regions or not it tends to schedule conservatively as already discussed in section 5.1. A lot of effort has been spent on loop optimizations on IPF. The classical examples are loops of the following nature:

```
void sum_scale (double* restrict x, double* restrict y, double* restrict z,
double S, long N) {
    long i;
    for (i=0; i<N;i++) z[i] = x[i] + y[i]*S ;
}
```

When compiled with `gcc -O3 -Drestrict=`, i.e. removing the “restrict” definition from the routine, the loop is scheduled in 11 cycles on Itanium 2 as illustrated here:

```
.b1_11:
{
    .mmi
    ldfd  f9=[r14],8 // cycle 0
    ldfd  f7=[r11],8
    add   r9=24,r10 ;;
} {
    .mii
    lfetch.nt1 [r10]
    mov   r10=r8
    mov   r8=r3
} {
    .mfb
    mov   r3=r9 ;;
} {
    .mfi
    fma.d f6=f7,f8,f9 // cycle 6
    nop.i 0 ;;
} {
    .mib
    stfd  [r2]=f6,8 // cycle 10
    br.ctop.sptk .b1_11 ;;
}
```

When compiled with `gcc -O3 -restrict`, i.e. honoring the “restrict” keyword, which testifies that the pointers do not alias, the compiler can use software pipelining and schedules the loop in only 2 cycles:

```
.b1_3:
{
    .mmi
    (p16) ldfd  f32=[r8],8 // cycle 0
    (p16) ldfd  f37=[r3],8 ;;
} {
    .mfi
    (p24) stfd  [r2]=f46,8 // cycle 1
    (p20) fma.d f42=f36,f8,f41
    (p16) add   r32=24,r35
} {
    .mib
    (p16) lfetch.nt1 [r35]
    br.ctop.sptk .b1_3 ;;
}
```

Although this example uses a loop to demonstrate the negative effects of ambiguous pointers, the problem can equally well occur in straight-line code.