



The University of Reading

Approaches for code optimisation on IA32

[using C and GNU/Linux]

Kai Uhlemann

Transferable Skills
Mrs. Nia Alexandrov

4. April 2005

Contents

Contents

1 Abbreviations	3
2 Abstract	4
3 Introduction	4
4 Main body	5
4.1 A Programmers Problem by Example	5
4.2 Solution in C	5
4.2.1 First Approach	5
4.2.2 Compiler Optimisation	6
4.2.3 Profiling the Code	8
4.2.4 Optimizing using Compiler Options	9
4.3 Optimizing using Intrinsic	10
4.4 Optimizing using Assembly	12
5 Results	15
5.1 Summary	15
5.2 Conclusion	15
A Appendix	17
A.1 Source Code scalar.c	17
References	21

1 Abbreviations

IA32	Intel 32bit architecture for microprocessors compatible to Intel 80386
GNU	GNU is not UNIX
GCC	GNU Compiler Collection
STDOUT	Standard output
CPU	Central Processing Unit
SIMD	Single Instruction Multiple Data
SSE	in fact ISSE, Internet Streaming SIMD Extensions

2 Abstract

This report gives an overview of some options you can use to optimise your C-Code working with GNU/Linux. It also will give a short introduction to the GCC C compiler as well as to common tools which assist you while optimising.

3 Introduction

It is a well known problem to many programmers in the world, that after you have implemented your code, you have to face, that your code does not work as fast as expected or needed.

What to do now? It is always a good idea to check the used algorithms and possibly find another one which suits your runtime needs. If the speed problem of your code is not too extensive, you might save time just by optimising your source code to the hardware platform you are currently using.

Within this report you will find some general ideas and approaches of how to adjust your code, to work faster.

First it is shown, what the common GCC can do for you, to increase the speed of a program. From there on this report will give you an introduction of the optimisation possibilities of "Intrinsics" and inline assembly.

To guide the reader, a program example is used to show the effectiveness of the used optimisation methods including some benchmarks.

Since the topic of this report is quite specific constricted to C programming on Linux systems using IA32, it is addressed to programmers who are familiar to these concepts.

This report is based on my research at the University of Applied Sciences in Berlin for "Parallel architectures" with my fellow student Karsten Reitz.

4 Main body

4.1 A Programmers Problem by Example

The programming example used for this report is a well known mathematical problem. The program calculates the scalar product of two vectors.

This example provides similar usage of arithmetic calculation on a huge amount of data.

A vector product calculates out of 2 vectors \vec{a} and \vec{b} with n elements each. The product is calculated as follows:

$$\vec{a} \cdot \vec{b} = (a_1 \dots a_n) \cdot \begin{pmatrix} b_1 \\ \dots \\ b_n \end{pmatrix} := a_1 b_1 + \dots + a_n b_n = \sum_{n=1}^n a_i b_i$$

4.2 Solution in C

4.2.1 First Approach

The following C routine calculates the scalar product exactly as the formula instructs. To get the scalar product, the routine adds all the products of the elements of both vectors.

To solve the problem for two n -length vectors about n multiplication and $n-1$ additions have to be done by the C program.

```
1 /* Computes the result by the use of plain c */
   int c_product(short int x[], short int y[])
   {
       short int temp=0;
6    register int index=0; /* use of prefix "register" speeds up */

       /* calculate the result in a simple loop */
       for (index=3; index<dim; index+=4)
       {
11    temp=temp+x[index-3]*y[index-3]+x[index-2]*y[index-2]\
        +x[index-1]*y[index-1]+x[index]*y[index];
```

```
}  
    return temp;  
}
```

The source can now be compiled using the GCC C compiler. For this report the gcc Version 3.35 was used. To compile the source code the following commands can be used:

```
user@unix: $ gcc -o scalar scalar.c
```

Now the program is ready for execution. For a very huge amount of data to be calculated, the program now might not be fast enough.

By default the used instruction to compile the code, the GCC C compiler does not make any changes or optimisations to improve the speed of the running program. That is the same like using `-O0` as additional Option while compiling. [GCCdoc]

So its obvious that the code can be optimised quite easily i.e. by using the optimisation capabilities of the GCC C compiler.

4.2.2 Compiler Optimisation

Following the online documentation related to the GCC there are plenty of possibilities how you can let the compiler optimise your code.

Optimization is activated using the `-O` switch followed by the level of optimization.

option	effect
-O0	This deactivates all optimization and is used by <i>default</i> .
-O1	1st level of optimization, takes more time to compile and may consume a lot more memory in larger functions and activates some optimization flags (for further details see the online manual of the GCC).
-O2	2nd stage of optimization, all option of -O1 are included as well as virtually every flag with does not tradoff speed for binary space. Note! Loop unrolling and function inlining are not included. This level increases the performance of your program as well as the time it needs to be compiled.
-O3	3rd level of optimization. Includes every option from -O2 and adds <i>-finline-functions</i> and <i>-frename-registers</i> options.
-Os	This level optimises for binary size and activates options to do so. This option activates all -O2 optimizations, which are not known to increase code size.

Table 1: Compiler optimization options [GCCdoc]

4.2.3 Profiling the Code

Before using the compiler optimization, it is necessary to know how to evaluate a program or just routines out of it.

There are a lot of tools familiar to the UNIX styled systems like i.e. the simple *time* which gives you the runtime of a program from start to end.

To check the execution time of routines and function within a program something more sophisticated is needed. *gprof* provides this and a lot more functionality which can be used to assess the programming example.

To use *gprof* the code has to be compiled by adding the *-pg* switch while compiling. By doing so, the compiler adds additional code for the profiling and time measurement.

So the command for compiling the C source code changes as follows:

```
user@unix: $ gcc -pg -o scalar scalar.c
```

While execution the profiling data is written to the file *gmon.out*, which is used after execution by *gprof* to analyse the program.

That not all the information which are collected (like the whole call graph) are of interest, the following command is used to get the runtime information using *gprof*:

```
user@unix: $ gprof -no-grpah -brief scalar gmon.out
```

This should return something like the following to the STDOUT:

Flat profile:

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
51.62	1.70	1.70	1	1.70	1.70	c_product

[..]

As you can see the time to execute the C routine was approximately 1.7 seconds.

4.2.4 Optimizing using Compiler Options

Besides the optimization via the `-O` switch it is also possible to let the compiler generate binary version specialized for the abilities of the machine it is running on.

While the test runs for this report were done on a Intel Xeon machine, the additional optimization flags are the following:

- `-march=pentium4`
- `-mfpmath=sse`

The `pentium4` was chosen due to the lack on a special Xeon architecture but the `pentium4` is the most related possible CPU type. With `-mfpmath=sse` the compiler generates floating point arithmetics for the SSE unit. [GCCx86]

To evaluate the effects of the different optimization options, it is necessary to create multiple version with different option deactivated and activated.

The following results were collected mixing the general compiler optimization via the `-O` switches in combination with the option for the architecture.

The execution times as shown above were determined using *gprof*. To prevent adulteration caused by concurrent running processes on the testing machine the program was executed 250 times. Only the fastest execution times are accounted for the chart above.

As you can see the optimization using `-O1` had nearly no speedup effect, only `-O2` and `-O3` had an observable effect.

But the execute time decreased only from 1.28s (`-O0` and `-mfpmath=sse` activated) to 1.00s (`-O2` or `-O2` and with or without `-mfpmath=sse` activated), that are just approximately 22% performance improvement.

If that improvement suits you, you may stop here, but there are ways to increase the performance even more.

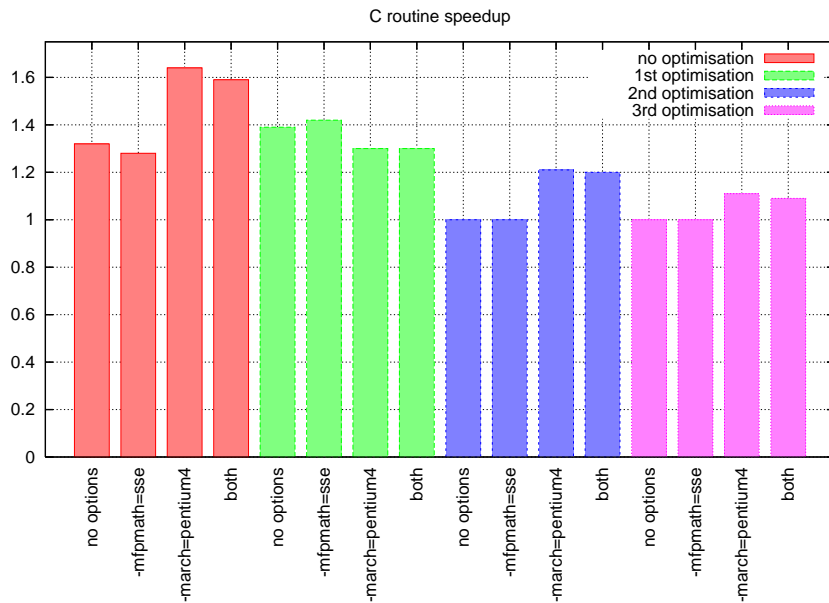


Figure 1: results C routine

4.3 Optimizing using Intrinsics

Since the usage of the SSE units through the compiler option did not work as well as expected, there are ways which the GCC provides to use special functions which make explicit usage of the SSE units. These special functions are called intrinsics.

Intrinsics encapsulate assembly routines in C functions, and are directly embedded into the source code via special header files. [LinuxMag03, page 68]

- *mmintrin.h* for MMX
- *xmmintrin.h* for SSE
- *emmintrin.h* for SSE2

Note! GCC supports since version 3.1 intrinsics for *MMX* and *SSE*, *SSE2* since version 3.3.

To use the intrinsics it is also necessary to add a switch while compiling. Via `-msse2` the functionalities are enabled.

```
user@unix: $ gcc -msse2 -pg -o scalar scalar.c
```

To test the effect of the usage of intrinsic in comparison to the C routine, an intrinsic routine as follows was created. This example uses SSE2, which provides arithmetic operation for double precision floating point values. [Intel]

```
/* Compute scalar product by the use of Intrinsics */
int intrin_product(short int x[], short int y[])
{
    __m128i *vecX = (__m128i *)x;
5   __m128i *vecY = (__m128i *)y;
    vector zwischen, temp;
    register int index=0;
    /*if there a better way for initialising, I don't know it so tell me*/
    zwischen.f[0]=zwischen.f[1]=zwischen.f[2]=zwischen.f[3]=
10   zwischen.f[4]=zwischen.f[5]=zwischen.f[6]=zwischen.f[7]=0;

    for (index=0;index<dim/8;index++)
    {
        /*multiply the vectors in 8 times steps*/
15   temp.m = _mm_mullo_epi16(vecX[index],vecY[index]);
        /*now adds the 8 short int values with the previous*/
        zwischen.m=_mm_add_epi16(zwischen.m,temp.m);
    }
    /*at the end, add all results to one and return to main*/
20   return (zwischen.f[0]+zwischen.f[1]+zwischen.f[2]+zwischen.f[3]+
        zwischen.f[4]+zwischen.f[5]+zwischen.f[6]+zwischen.f[7]);
}
```

The following results were collected using intrinsics.

The reason, why the execution times using `-O0` are much longer than the rest, is caused by the implementation of the intrinsic function as inline function. Since `-O0` prevents this, a lot of time is wasted by calling the intrinsic routines.

Since the intrinsics are quite specially optimised to the hardware, the general optimization switches cannot improve the execution time any more. The fastest execution measured was 0.27s. With exception of the programs compiled with `-O0` all the measured times were all about the same.

In comparison to the C routine which runs fastest approximately 1s, you can get an

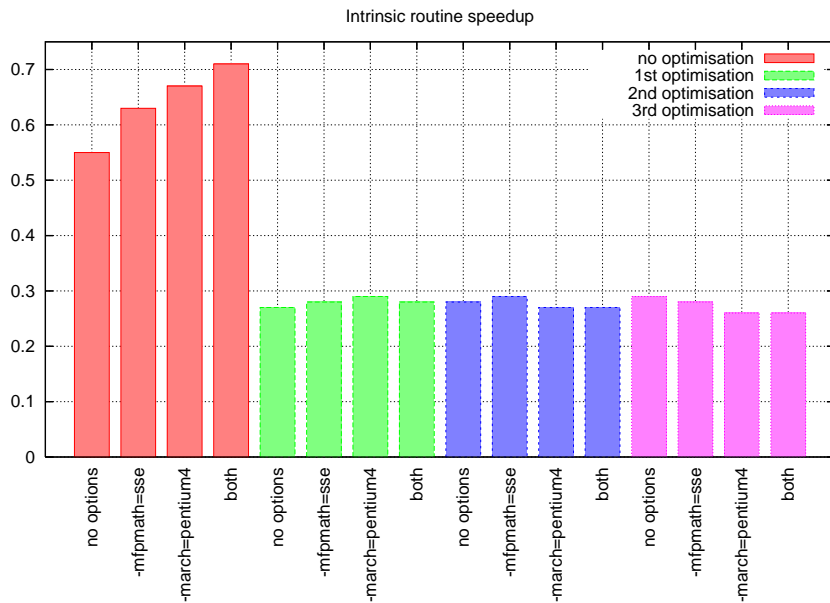


Figure 2: results intrinsics

increase of 73% of the program performance.

4.4 Optimizing using Assembly

The results using intrinsics were quite outstanding. But can the performance be increased even more by using inline assembly? The GCC uses the AT&T syntax for assembly.

This might be the most challenging part because most programmers using Intel IA32 are used to the Intel syntax.

siehe Tabelle 2

An embedded assembly block in C looks as the following [IBM01]:

C-Code...

```
__asm( assembler template
```

Name	Intel	AT&T
registers	eax	%%eax
source & destination order	destination,source	source,destination
immediate,constant	_constant	\$_constant
dataformat for operations	mov bx,ax	movw %%ax,%%bx

Table 2: important differences between Intel and AT&T syntax

```

:output operands(optional)
:input operands (optional)
list of clobbered registers(optional)
);

```

... C-Code

A good start to get used to AT&T syntax is to visit [IBM01] were the information from above is taken.

The following source code shows how to use inline assembly to use the SSE2 units to calculate the scalar product:

```

/* Compute scalar product by the use of inline assembler */
int asm_product(short int x[], short int y[])
3 {
    short int temp[]={0,0,0,0,0,0,0,0};
    int i;
    /* clear xmm2 register */
    asm volatile (
8     "pxor %%xmm2,%%xmm2;" ::);

    /* calculate the products and summ them up in xmm2*/

13    for(i = 0; i < dim; i += 8){
        asm volatile (
            "movdqu %0,%%xmm0;" //move quad word from x vector into xmm0 register
            "movdqu %1,%%xmm1;" //move quad word from y vector into xmm1 register
            "pmullw %%xmm1,%%xmm0;" //packed multiply xmm1 and xmm0
18         "paddw %%xmm0,%%xmm2;" // add the product to xmm2
            :
            : "m" (x[i]), "m" (y[i]));
        }

23    asm volatile (
        "movdqu %%xmm2, %0;" //move result vector to temp
        "emms" //reset mm register for any following fp operations
        : "=m" (temp[0]) //use temp as output

```

4 Main body

```
    :);  
28     return temp[0]+temp[1]+temp[2]+temp[3]+temp[4]+temp[5]+temp[6]+temp[7];  
    }
```

The following results were collected using assembly.

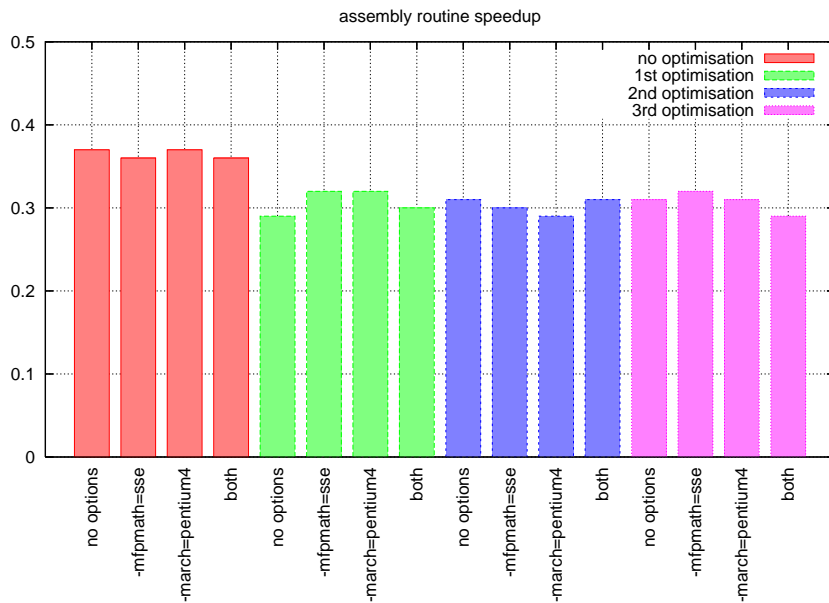


Figure 3: results assembly

The results shown above prove the excellent work of the intrinsic used in the previous routine. The fastest measured execution using assembly was 0.29s which is as fast as the intrinsic version.

The usage of intrinsics here shows how effective and less extensive it can be in comparison to the speedup you will get.

5 Results

5.1 Summery

As shown in the report, there are some general ideas of how you can improve the execution of your program.

The first would be to let the compiler try to optimise you code by using optimisation switches. If that does not provide you with the necessary speedup you need, you have to modify your code manually.

Intrinsics are a good way to improve the source code manually without knowing too much technical details about the platform the program is running on. Intrinsics are like assembly optimisation without the knowlegde of any assembly syntax.

Since the intrinsics provide a lot of function which can be used, they are somewhat limited in the range of optimisation they can provide. If you face serious performance problems, it might be the best way to use assembly code embedded to your code.

5.2 Conclusion

As you could see the optimisations made by the compiler are not extraordinary, but a good start to speed up your program. Only the last 2 optimisation switches (`-O2` and `-O3` increase the performance enough to be mentioned.

The specific hardware optmisation switches (`-march=pentium4` and `-mfpmath=sse`) could not prove any speedup for the C program. Especially the first switch increased the execution time in comparison to the version compiled without hardware specific options activated. This is probably because the architecture of the Pentium4 und Xeon are slightly different.

Nevertheless the benchmarks show, that intrinsics and embedded assembly code are quite sophisticated to guarantee a speedup of the example program of up to 73%.

Especially the intrinsics are a simple to use and very easy to handle tool to improve the execution time of a program.

A Appendix

A.1 Source Code scalar.c

```

/*****
 * Copyright (C) 2004 by Karsten Reitz and Kai Uhlemann *
 *
 * This program is free software; you can redistribute it and/or modify *
5 * it under the terms of the GNU General Public License as published by *
 * the Free Software Foundation; either version 2 of the License, or *
 * (at your option) any later version. *
 *
 * This program is distributed in the hope that it will be useful, *
10 * but WITHOUT ANY WARRANTY; without even the implied warranty of *
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the *
 * GNU General Public License for more details. *
 *
 * You should have received a copy of the GNU General Public License *
15 * along with this program; if not, write to the *
 * Free Software Foundation, Inc., *
 * 59 Temple Place – Suite 330, Boston, MA 02111–1307, USA. *
*****/

20
// Include SIMD Header
#ifdef __INTEL_COMPILER
# include <dvec.h> // also contains xmmintrin.h
#else
25 # include <emmintrin.h>
#endif

#include <stdio.h>
#include <stdlib.h>
30 #include <errno.h>
#include <string.h>
#define FREE( x ) { free( x ); x = NULL; }
#define dim 1024*1024
#define dim_alloc 10
35 #define value 1
#define value2 -1

typedef union{
40 __m128i m; short int f[8];
} vector;

int intrin_product(short int x[], short int y[]);
int c_product(short int x[], short int y[]);
45 int asm_product(short int x[], short int y[]);
void fillvec(short int *var, short int even);

```

A Appendix

```
int main(void)
{
50   short int vector_a[dim],vector_b[dim];
      int scpro=0;

      /*first fill both vectors with data*/
55   fillvec(vector_a,0);
      fillvec(vector_b,1);

      /*now multiply the vectors*/
      scpro=intrin_product(vector_a, vector_b);
60   printf(" intrinsic result= %d\t",scpro);

      scpro=c_product(vector_a, vector_b);
      printf("c result= %d\t",scpro);

65   scpro=asm_product(vector_a, vector_b);
      printf("asm result= %d\n",scpro);

      return 0;
}
70
/* Compute scalar product by the use of inline assembler */
int asm_product(short int x[], short int y[])
{
      short int temp[]={0,0,0,0,0,0,0,0};
75   int i;
      /* clear xmm2 register */
      asm volatile (
          "pxor %%xmm2,%%xmm2;"::);

80
          /* calculate the products and summ them up in xmm2*/

          for(i = 0; i < dim; i += 8){
              asm volatile (
85   "movdqu %0,%%xmm0;" //move quad word from x vector into xmm0 register
          "movdqu %1,%%xmm1;" //move quad word from y vector into xmm1 register
          "pmullw %%xmm1,%%xmm0;"//packed multiply xmm1 and xmm0
          "paddw %%xmm0,%%xmm2;" // add the product to xmm2
          :
90   : "m" (x[i]), "m" (y[i]));
          }

          asm volatile (
95   "movdqu %%xmm2, %0;" //move result vector to temp
          "emms" //reset mm register for any following fp operations
          :="m" (temp[0]) //use temp as output
          :);
```

A Appendix

```
    return temp[0]+temp[1]+temp[2]+temp[3]+temp[4]+temp[5]+temp[6]+temp[7];
100 }

/* Compute scalar product by the use of Intrinsic */
int intrin_product(short int x[], short int y[])
{
105  __m128i *vecX = (__m128i *)x;
    __m128i *vecY = (__m128i *)y;
    vector zwischen,temp;
    register int index=0;
    /*if there a better way for initialising , I don't know it so tell me*/
110  zwischen.f[0]=zwischen.f[1]=zwischen.f[2]=zwischen.f[3]=
        zwischen.f[4]=zwischen.f[5]=zwischen.f[6]=zwischen.f[7]=0;

    for (index=0;index<dim/8;index++)
    {
115     /*multiply the vectors in 8 times steps*/
        temp.m = _mm_mullo_epi16(vecX[index],vecY[index]);
        /*now adds the 8 short int values with the previous*/
        zwischen.m=_mm_add_epi16(zwischen.m,temp.m);
    }
120  /*at the end, add all results to one and return to main*/
    return (zwischen.f[0]+zwischen.f[1]+zwischen.f[2]+zwischen.f[3]+
        zwischen.f[4]+zwischen.f[5]+zwischen.f[6]+zwischen.f[7]);
}

125 /* Computes the result by the use of plain c */
int c_product(short int x[], short int y[])
{
    short int temp=0;
130  int index=0; /* use of prefix "register" speeds up */

    /* calculate the result in a simple loop */
    for (index=7;index<dim;index+=8)
    {
135     temp=temp+x[index-7]*y[index-7]+x[index-6]*y[index-6]+x[index-5]*y[index-5]
        +x[index-4]*y[index-4]+x[index-3]*y[index-3]+x[index-2]*y[index-2]
        +x[index-1]*y[index-1]+x[index]*y[index];
    }
    return temp;
140 }

/* Fills the vector with values */
void fillvec(short int *var, short int even)
145 {
    int index=0;
    if(even==0){
        for (index=0;index<dim;index+=2)
```

A Appendix

```
150  {
    var [index]= value;
    var [index+1]= value;
  }
  }
  else
155  {
  for (index=0;index<dim; index+=2)
  {
    var [index]= value;
    var [index+1]= value2;
160  }
  }
}
```

List of Figures

1	results C routine	10
2	results intrinsics	12
3	results assembly	14

List of Tables

1	Compiler optimization options [GCCdoc]	7
2	important differences between Intel and AT&T syntax	13

References

[GCCdoc] Optimize Options - Using the GNU Compiler Collection (GCC).<http://gcc.gnu.org/onlinedocs/gcc-3.3.5/gcc/Optimize-Options.html#Optimize-Options>

[GCCx86] i386 and x86-64 Options - Using the GNU Compiler Collection (GCC).http://gcc.gnu.org/onlinedocs/gcc-3.3.5/gcc/i386-and-x86_002d64-Options.html#i386-and-x86_002d64-Options

[LinuxMag03] Stephan Siemen (2003):Linux Magazine September 2003. Linux New Media AG München/Germany

[Intel] Intel Corporation:Intel® C++ Compiler User's Guide. Online version.http://www.intel.com/software/products/compilers/clin/docs/ug_cpp/

[IBM01] Bharata B. Rao (2001):Inline assembly for x86 in Linux. PDF-Version.IBM Developer Works. <http://www-106.ibm.com/developerworks/library/l-ia.html>