Introduction to Function Multiversioning (FMV) in GCC 6

While new architecture provides many powerful instruction set extensions, it is challenging for developers to generate code that takes advantage of these capabilities. Current Linux* developers can't always take advantages of all new architectures coming to the market every year without losing backward compatibility. For example, for math-heavy code one may want to turn on AVX, while also wanting the binary to run on earlier CPUs. This problem stops users from benefiting from the best of the new computing architecture technology, from low-level kernel features to complex applications.

Despite the fact that newer versions of GCC and kernel try to provide the tools to use the new architecture features before the platforms appear in the market, is not always easy for developers to use them. Currently, C code developers have few choices to solve this problem:

- Write multiple versions of their code, each targeting different instruction set extensions, and manually handle runtime dispatching of these versions
- Generate multiple versions of their binary, each targeting a different platform
- Choose a minimum hardware requirement that will not take advantage of newer platforms

Despite these challenges, the benefit of using the new architecture technology is compelling. For example, consider the second version of the advanced vector extension instruction set extension (AVX2), introduced in the 4th Generation Intel® Core™ processor family (formerly known as Haswell). The benefits of using this technology in scientific computing fields are well understood. Such is the case of the Basic Linear Algebra libraries (OPENBLAS). The use of AVX2 in OPENBLAS libraries gave projects like R language a boost in performance of up to 2x in execution time.

However, this means a lot of work in terms of development, deployment and manageability in the long term. The idea of having multiple versions of their binaries, one for each architecture, discourages the developers and OS distributions to support these features. With the advances in computing architecture growing every year, this seems like a herculean task.

It would be better to optimize some key hot functions for multiple architectures and execute them when the binary detects the CPU capabilities at runtime. This feature exists in GCC 5 only for C++ and is known as Function Multi Versioning (FMV). FMV in GCC5 make it easy for the developer to specify multiple versions of a function, each catered to a specific target ISA feature. GCC then takes care of creating the dispatching code necessary to execute the right function version.

If you want to use FMV in your C++ code you may specify multiple versions of a function. For example:

```
__attribute__ ((target ("default")))
  int foo ()
  {
```

```
// The default version of foo.
return 0;
}

__attribute__ ((target ("sse4.2")))
int foo ()
{
// foo version for SSE4.2
  return 1;
}

__attribute__ ((target ("arch=atom")))
int foo ()
{
// foo version for the Intel ATOM processor
return 2;
}

__attribute__ ((target ("arch=amdfam10")))
int foo ()
{
// foo version for the AMD Family 0x10 processors.
return 3;
}

int main ()
{
int (*p)() = &foo;
assert ((*p) () == foo ());
```

```
	return 0;
	}
```

On the other hand, the incoming GCC 6 will support Function Multi Versioning in C code, making it easier to develop Linux applications that take advantage of the enhanced instructions of the new architectures. A simple example where FMV can take advantage of the AVX technology could be represented with an arrays addition:

```
#define MAX 1000000
int a[256], b[256], c[256];


__attribute__((target_clones("arch=core-avx2","arch=atom","arch=slm","defa
ult")))
void foo(){
	int i,x;
	for (x=0; x<MAX; x++){
		for (i=0; i<256; i++){
			a[i] = b[i] + c[i];
		}
	}
 }
int main () {
	foo();
	return 0;
}
```

As we can see, the selection of the supported architecture is pretty simple. The developer just needs to select the minimum set of architectures to support, such as: AVX2, Intel® Atom™, AMD* or SLM. At the end, the object dump of this code will have the optimized assemble instructions for each architecture,for example:

```
Basic code :
add    %eax,%edx
```

```
Vectorized code :

vmovdqa 0x0(%rax),%xmm0
vpaddd 0x0(%rax),%xmm0,%xmm0
vmovaps %xmm0,0x0(%rax)

AVX2 code :

vmovdqu (%r8,%rax,1),%ymm0
vpaddd (%r9,%rax,1),%ymm0,%ymm0
vmovdqu %ymm0,(%rdi,%rax,1)
```

Normally, telling the compiler to use AVX2 instructions would limit our binary to Haswell and newer processors. With FMV, the compiler can generate AVX-optimized versions of the code and will automatically, at runtime, ensure that only the appropriate versions are used. In other words, when the binary is run on Haswell or later generation CPUs, it will use Haswell-specific optimizations, and when that same binary is run on a pre-Haswell generation processor, it will fall back to using the standard instructions supported by the older processor. The object dump will have the three kinds of instructions sets and registers available.

CPUD selection

In GCC5 version of FMV there is a dispatch priority. The order in which the versions should be dispatched is prioritized to each function version based on the target attributes. Function versions with more advanced features get higher priority. For example a version targeted for AVX2 will have a higher dispatch priority than a version targeted for SSE2.

Memory footprint

The increment in memory footprint is not a problem. In current experiments the maximum overhead in memory footprint has been up to 30% of the original size. Considering the current workstations and resource capabilities of data centers,, this increment in memory footprint is not a critical concern.

Results

The following table shows that on a Haswell platform, the FMV version of the binary executes with the same performance as a binary build with just AVX2 instructions. On older processors, the same binary would still work, but the performance would be lower due to the lack of optimized instructions.

GCC flags	Execution Time
gcc sanity.c -o sanity	603 ms
gcc -O3 sanity.c -o sanity	38 ms
gcc -O3 sanity.c -mavx2 -o sanity	26 ms
gcc -O3 sanity-fmv.c -o sanity-fmv	25 ms

Real world example

Today more and more industry segments are benefiting from the use of cloud scientific computing. These segments include chemical engineering, digital content creation, and financial service and analytics applications among others. As we can see, the importance of improving the performance of scientific computing libraries is mandatory for a cloud OS.
One of these scientific computing libraries is the NumPy library. NumPy is the fundamental package for scientific computing with Python. Among other things, it includes support for large, multidimensional arrays and matrices. This library has special features for linear algebra, Fourier transform, and random number capabilities among others.

The advantages of using FMV technology in a scientific library as NumPy is well-understood and accepted. If vectorization is not enabled, there will be a lot of unused space in SIMD registers. If vectorization is enabled, the compiler may use the additional registers to perform more operations (in our example the additions of more integers) in a single instruction. As we can see in the following table, the performance boost thanks to the FMV technology (running in a Haswell machine with AVX2 instructions) can be up to 3% in terms of execution time for scientific workloads.

GCC flags	Execution Time
Numpy with -O3	8600 seconds
Numpy with FMV patches	8400 seconds

Next steps

The Clear Linux Project for Intel Architecture is focusing on applying FMV technology on packages where it is detected that AVX2 instructions can give an improvement. In order to solve the problems

of supporting FMV in a full Linux distribution, Clear Linux Project for Intel Architecture provides a patch generator based on vectorization-candidate detection (gcc flag `-fopt-info-vec)`. This tool can provide all the FMV patches that a Linux distribution might use. We are selecting the ones that gave a significant performance improvement based on our benchmarks. There are other compiler optimization techniques that take advantage of the profile data to perform additional optimizations based on how the code behaves. Clear Linux Project for Intel Architecture will use these profiling features as well as the FMV approach to improve the performance of user applications as much as possible.