Double precision floating point alignment issue
Wen X.

The effect of data alignment on performance is a well-known issue. In particular, for an IA-32 system the data body are aligned to 4-byte boundaries by default. This means that data units larger than 4 bytes, such as double-precision floating points, might be misaligned, which can bring substantial penalty on the throughput of related instructions.

This article addresses the alignment issue of double-precision floating point numbers in C. The solution provided here also applies to other 8-byte data bodies, and can be extended easily to longer (s.a. 16-byte) ones.

1. The heap

Dynamic memories are allocated from the heap. When a memory block is allocated from the heap, the default data alignment depends on the allocation function (such as malloc or new), which is likely to offer 4-byte alignment in a 32-bit environment. According 8-byte data bodies may be misaligned.

The following malloc8(…) and free8(…) functions, when used instead of malloc(…) and free(…), guarantee 8-byte alignment of returned pointer.

```
void* malloc8(unsigned size)
{
 char *buffer, *result;
 buffer=(char*)malloc(size+8+sizeof(void*));
 if(!buffer) return(NULL);
 char* tmp=&buffer[sizeof(void*)];
 result=&((char*)((unsigned)tmp&0xFFFFFFF8))[8];
 ((void**)result)[-1]=buffer;
 return(result);
}

void free8(void* buffer8)
{
 free(((void**)buffer8)[-1]);
}
```

What this piece of code does is to allocate a slightly larger *buffer, containing enough memory for the desired block, plus sizeof(void*) to store the buffer pointer itself, and a 8-byte slot for the purpose of alignment. The returned pointer is then directed to the first 8-byte aligned position that leaves at least sizeof(void*) bytes before it. The sizeof(void*) bytes immediately before the returned pointer is used to store the "true" buffer pointer, which is freed when free8(…) is called. The returned pointer cannot be freed using the normal free(…) function.

2. The stack

Local variables are allocated from the stack intrinsically during the function all. First the arguments are pushed onto the stack in the order specified by the calling convention (_cdecl, _pascal, etc), then the call instruction is executed. Local variables are then allocated from where the current stack pointer (ESP) points at by further deducing ESP. Generally the stack pointer before this deduction is stored to the base pointer register (EBP), which is then used to refer to all local variables, including arguments as [EBP+] and normal local variables, including register-passed variables when _fastcall convention is used, as [EBP-]. In an IA-32 environment ESP is 4-byte aligned by default. Our goal is to align all local 8-byte data bodies involved in arithmetic, including arguments and local variables.

2.1 Stack alignment by caller function

The following macro pair, applied immediately before and after a function call, aligns the stack point to an 8-byte boundary.

```
#define ALIGH_STACK_8 {int alignstack; asm {mov alignstack, esp} \
        asm {and esp, 0xFFFFFFF8}

#define RESTORE_STACK asm {mov esp, alignstack} }
```

ALIGN_STACK_8 stores ESP to a local variable then align ESP to the nearest 8-byte boundary in the deducing direction. RESTORE_STACK recovers the aligned ESP to its stored value.

ALIGN_STACK_8 guarantees that the first pushed argument is 8-byte aligned. The programmer maintains the alignment of individual argument by carefully arranging their order, so that the number of 4-byte bodies pushed before pushing any 8-byte body is always even.

The local variables are generally allocated in the order of declaration, yet there is no 100% guarantee that all compilers comply to it. However, it is very likely that 8-byte bodies declared together without any other declarations in between will be aligned among themselves. Their common alignment can usually be controlled by repositioning a 4-byte variable declaration. (And if this fails we can always declare one more 4-byte argument to change the stack top of local variables.) If the way the compiler allocates local variables is known, we can safely count out the declarations to make sure the 8-byte bodies are aligned. However, an easier way is to simply run the program and check the alignment, and rearrange the declarations to align the variables if misalignment happens. By doing this we also bypass the difficulty introduced by possible far calls, which pushes one more pointer executing the call instruction.

Stack alignment by caller function has minimal impact on the C code, since the alignment is finished before anything related to the function call happens.

2.2 Stack alignment by callee function

Usually stack alignment are desired only for a limited number functions doing heavy floating-point computation in double precision, which are called a lot. Although the caller-function-aligns-stack scheme offers a clean and minimal-impact solution, it requires each function call be enclosed by two macros. It not only makes the caller code cumbersome, but also leaves an opportunity of alignment failure due to the forgetfulness of our dear programmer. From the function user's view it is desirable to keep stack alignment work invisible. To do this we need something like

```
int function(int arg1, double arg2, …)
{
   ALIGN_I_8
      <the function body>
   RESTORE_I
      return result;
}
```

The difficulty is that once the function is called, all the local variables, including the arguments, are accessed through EBP. This implies:

1. The alignment macros must change EBP *and* ESP to solve misalignment;

2. The same variable may indicate different physical memories inside and outside the macro pair;

3. The arguments cannot be accessed directly from inside the macro pair; local variables whose values are acquired inside the macro pair cannot be accessed directly from outside the macro pair.

The following macro pair aligns EBP to an 8-byte boundary

```
#define ALIGH_I_8 {asm {mov ecx, ebp} asm {and ebp, 0xFFFFFFF8} \
          asm {sub ecx, ebp} asm {sub esp, ecx} asm {push ecx}

#define RESTORE_I asm {pop ecx} asm {add esp, ecx} asm {add ebp, ecx}}
```

The stack top stores the amount of shift done to EBP. This value is necessary to access arguments from the function body, or to access local variables from outside the macro pair (for return value). This type of cross-access is not only hard to read (probably in assembly), but also slow to execute. Therefore I suggest this macro pair be used only when there are no or very few such cross-accesses. Otherwise the caller-aligns-stack method is preferred with the following wrapper:

```
int function(int arg1, double arg2, …)
{
   int result;
   ALIGN_STACK_8
      result=function_unaligned(int arg1, double arg2, …)
   RESTORE_STACK
      return result;
}
```

This wrapper also allows the argument be arranged user-friendly regardless of the alignment issue.