Index Nav: [Date Index] [Subject Index] [Author Index] [Thread Index]
Message Nav: [Date Prev] [Date Next] [Thread Prev] [Thread Next]
Other format: [Raw text]

A proposal to align GCC stack

- *From*: "Ye, Joey" <joey dot ye at intel dot com>
- *To*: <gcc at gcc dot gnu dot org>
- *Cc*: "H.J. Lu" <hjl at lucon dot org>, <xuepeng dot guo at intel dot com>
- *Date*: Tue, 18 Dec 2007 10:25:42 +0800
- *Subject*: A proposal to align GCC stack

-- 0. MOTIVATION --
Some local variables (such as of __m128 type or marked with alignment attribute) require stack aligned at a boundary larger than the default stack boundary. Current GCC partially supports this with limitations. We are proposing a new design to fully solve the problem.

-- 1. CURRENT IMPLEMENTATION --
There are two ways current GCC supports bigger than default stack alignment. One is to make sure that stack is aligned at program entry point, and then ensure that for each non-leaf function, its frame size is aligned. This approach doesn't work when linking with libs or objects compiled by other psABI confirming compilers. Some problems are logged as PR 33721. Another is to adjust stack alignment at the entry point of a function if it is marked with __attribute__ ((force_align_arg_pointer)) or -mstackrealign option is provided. This method guarantees the alignment in most of the cases but with following problems and limitations:

* Only 16 bytes alignment is supported
* Adjusting stack alignment at each function prologue hurts performance unnecessarily, because not all functions need bigger alignment. In fact, commonly only those functions which have SSE variables defined locally (either declared by the user or compiler generated internal temporary variables) need corresponding alignment.
* Doesn't support x86_64 for the cases when required stack alignment is > 16 bytes
* Emits inefficient and complicated prologue/epilogue code to adjust stack alignment
* Doesn't work with nested functions
* Has a bug handling register parameters, which resulted in a cpu2006 failure. A patch is available as a workaround.

-- 2. NEW PROPOSAL: DESIGN --
Here, we propose a new design to fully support stack alignment while overcoming above problems. The new design will
* Support arbitrary alignment value, including 4,8,16,32...
* Adjust function stack alignment only when necessary
* Initial development will be on i386 and x86_64, but can be extended to other platforms
* Emit more efficient prologue/epilogue code
* Coexist with special features like dynamic stack allocation (alloca),

nested functions, register parameter passing, PIC code and tail call optimization
* Be able to debug and unwind stack

2.1 Support arbitrary alignment value
Different source code and optimizations requires different stack alignment,
as in following table:

Feature	Alignment (bytes)
i386_ABI	4
x86_64_ABI	16
char	1
short	2
int	4
long	4/8*
long long	8
__m64	8
__m128	16
float	4
double	8
long double	4/16*
user specified	any power of 2

*Note: 4 for i386, 8/16 for x86_64
The new design will support any alignment value in this table.

2.2 Adjust function stack alignment only when necessary

Current GCC defines following macros related to stack alignment:
i. STACK_BOUNDARY in bits, which is enforced by hardware, 32 for i386 and 64 for x86_64. It is the minimum stack boundary. It is fixed.
ii. PREFERRED_STACK_BOUNDARY. It sets the stack alignment when calling a function. It may be set at command line and has no impact on stack alignment at function entry. This proposal requires PREFERRED >= STACK, and by default set to ABI_STACK_BOUNDARY

This design will define a few more macros, or concepts not explicitly defined in code:
iii. ABI_STACK_BOUNDARY in bits, which is the stack boundary specified by psABI, 32 for i386 and 128 for x86_64. ABI_STACK_BOUNDARY >= STACK_BOUNDARY. It is fixed for a given psABI.
iv. LOCAL_STACK_BOUNDARY in bits. Each function stack has its own stack alignment requirement, which depends the alignment of its stack variables,
LOCAL_STACK_BOUNDARY = MAX (alignment of each effective stack variable).
v. INCOMING_STACK_BOUNDARY in bits, which is the stack boundary at function entry. If a function is marked with __attribute__ ((force_align_arg_pointer)) or -mstackrealign option is provided, INCOMING = STACK_BOUNDARY. Otherwise, INCOMING == MIN(ABI_STACK_BOUNDARY, PREFERRED_STACK_BOUNDARY) because a function can be called via psABI externally or called locally with PREFERRED_STACK_BOUNDARY.
vi. REQUIRED_STACK_ALIGNMENT in bits, which is stack alignment required by local variables and calling other function. REQUIRED_STACK_ALIGNMENT == MAX(LOCAL_STACK_BOUNDARY,PREFERRED_STACK_BOUNDARY) in case of a non-leaf function. For a leaf function, REQUIRED_STACK_ALIGNMENT == LOCAL_STACK_BOUNDARY.

This proposal won't adjust stack when INCOMING_STACK_BOUNDARY >= REQUIRED_STACK_ALIGNMENT. Only when INCOMING_STACK_BOUNDARY < REQUIRED_STACK_ALIGNMENT, it will adjust stack to REQUIRED_STACK_ALIGNMENT at prologue.

2.3 Initial development on i386 and x86_64

We initially support i386 and x86_64. In this document we focus more on i386 because it is hard to implement because of the restriction of having
a small register file. But all that we discuss can be easily applied to x86_64.

2.4 Emit more efficient prologue/epilogue

When a function needs to adjust stack alignment and has no dynamic stack allocation, this design will generate following example prologue/epilogue
code:

```
IA32 example Prologue:
        pushl     %ebp
        movl      %esp, %ebp
        andl      $-16, %esp
        subl      $4, %esp ; is $-4 the local stack size?
Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        ret
```

Locals will be addressed as esp + offset and parameters as ebp + offset.

Add x86_64 example here.

Thus BP points to parameter frame and SP points to local frame.

2.5 Coexist with special features

Stack alignment adjustment will coexist with varying GCC features that have special calling conventions and frame layout, such as dynamic stack allocation (alloca), nested functions and parameter passing via registers to local functions.

I386 hard register usage is the major problem to make the proposal friendly
to various GCC features. This design requires an additional hard register
in prologue/epilogue in case of dynamic stack allocation. Because I386 PIC
requires BX as GOT pointer and I386 may use AX, DX and CX as parameter passing registers, there are limited candidates for this proposal to choose. Current proposal suggests EDI, because it won't conflict with i386 PIC or regparm.

X86_64 is much easier. This proposal just chooses RBX.

2.5.1 When stack alignment adjustment comes together with alloca, following
example prologue/epilogue will be emitted:

```
Prologue:
       pushl     %edi                      // Save callee save reg edi
       leal      8(%esp), %edi             // Save address of parameter
frame
       andl      $-16, %esp                // Align local stack

//  Reserve two stack slots and save return address
//  and previous frame pointer into them. By
//  pointing new ebp to them, we build a pseudo
//  stack for unwinding.
       pushl     $4(%edi)                  //  save return address
       pushl     %ebp                      //  save old ebp
       movl      %esp, %ebp                //  point ebp to pseudo frame
start

       subl      $24, %esp                 // adjust local frame size
       movl      %edi, vreg1

epilogue:
       movl      vreg1, %edi
       movl      %ebp, %esp              // Restore esp to pseudo frame
start
       popl      %ebp
       leal      -8(%edi), %esp          // restore esp to real frame
start
       popl      %edi                    // Restore edi
       ret
```

Locals will be addressed as ebp - offset, parameters as vreg1 + offset

Where BX is used to set up virtual parameter frame pointer, BP points to local frame and SP points to dynamic allocation frame.

2.5.2 Nested functions will automatically work because it uses CX as static pointer, which won't conflict with any registers used by stack alignment adjustment, even when nested functions are called via function pointer and a function stub on stack.

2.5.3 GCC may optimize to use registers to pass parameters . At most AX, DX and CX will be used. Such optimization won't conflict with stack alignment adjustment thus it should automatically work.

2.5.4 I386 PIC uses EBX as GOT pointer. This design work well under i386 PIC:

For example:

```
i686 Prologue:
        pushl     %edi
        leal      8(%esp), %edi
        andl      $-16, %esp
        pushl     $4(%edi)
        pushl     %ebp
        movl      %esp, %ebp
        subl      $24,  %esp
        call      .L1
.L1:
        popl      %ebx
        movl      %edi, vreg1

Body:  // code for alloca
        movl      (vreg1), %eax
        subl      %eax, %esp
        andl      $-16, %esp
        movl      %esp, %eax

i686 Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        leal      -8(%edi), %esp
        popl      %edi
        ret
```

Locals will be addressed as ebp - offset, parameters as vreg1 + offset, ebx has the GOT pointer.

2.6 Debug and unwind will work since DWARF2 has the flexibility to define different frame pointers.

2.7 Some intrinsics rely on stack layout. Need to handle them accordingly. They are __builtin_return_address, __builtin_frame_address. This proposal will setup pseudo frame slot to help unwinder find return address and parent frame address by emit following prologue code after adjusting alignment:

```
        pushl     $4(%edi)
        pushl     %ebp
```

-- 3. NEW PROPOSAL: IMPLEMENTATION --

The proposed implementation can be partitioned into following subtasks.

* Alignment requirement collection
* Frames addressing
* Alignment code generation
* Debug and unwind information

3.1 Collect alignment requirement

Collecting each function's alignment requirement from frontend or from optimization passes like vectorizer, and informing backend.

Current GCC uses cfun->stack_alignment_needed to store MIN(largest stack variable alignment, PREFERRED_STACK_BOUNDARY). We will reuse this field and
define its value only as "largest stack variable alignment"

3.2 Frames addressing

Adding parameter frame, local frame, static frame and dynamic frame with appropriate pointers, either hard registers or virtual registers.

Backend will customize CAN_ELIMINATE hook to assign hard registers to corresponding virtual registers.

3.3 Alignment code generation

Emit prologue/epilogue code to guarantee correct stack alignment based on
each function's alignment requirement collected previously.

Modification should happen in ix86_expand_prologue and ix86_expand_epilogue.
Code to be emitted can follow above design in a straight forward manner.

3.4 Debug information

Emit debug and unwind information for aligned stacks. It also happens in ix86_expand_prologue and ix86_expand_epilogue corresponding the prologue/epilogue code emitted.

4. Code Example

Simply function:

```
void foo()
{

   volatile int local;
   ...
}
```

```
i686 Prologue:
        pushl     %ebp
        movl      %esp, %ebp
        subl      $4, %esp         // Adjust local frame size by 4
i686 Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        ret



x86_64 Prologue:
        pushq     %rbp
        movq      %rsp, %rbp
        subq      $16, %rsp
x86_64 Epilogue:
        movl      %rbp, %rsp
        popl      %rbp
        ret

Pure 16 bytes align:
void foo()
{
    volatile __m128 m = _mm_set_ps1(0.f);
}

i686 Prologue:
        pushl     %ebp
        movl      %esp, %ebp
        andl      $-16, %esp
        subl      $16, %esp     // this is space for m, 16 byte aligned
i686 Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        ret

x86_64 Prologue:
        pushq     %rbp
        movq      %rsp, %rbp
        andq      $-16, %rsp
        subq      $16, %rsp
x86_64 Epilogue:
        movl      %rbp, %rsp
        popl      %rbp
        ret

16 bytes align with alloca:
void foo(int size)
{
    char * ptr=alloca(size);
    volatile int __attribute((aligned(32)) m = 0;
    ...
}

i686 Prologue:
        pushl     %edi
        leal      8(%esp), %edi
        andl      $-32, %esp
        pushl     $4(%edi)
        pushl     %ebp
        movl      %esp, %ebp
        subl      $24,  %esp

Body:  // code for alloca
        movl      %edi, vreg1
        movl      (vreg1), %eax
        subl      %eax, %esp
        andl      $-16, %esp
        movl      %esp, %eax

i686 Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        leal      -8(%edi), %esp
        popl      %edi
        ret

void foo(int dummy1, int dummy2, int dummy3, int dummy4,
         int dummy5, int dummy6, int size)
{
    char * ptr=alloca(size);
    volatile int __attribute((aligned(32)) m = 0;
    ...
}
x86_64 Prologue:
        pushq     %rbx
        leaq      $16(%rsp), %rbx
        andq      $-32, %rsp
        pushq     8(%rbx)
        pushq     %rbp
        movq      %rsp, %rbp
        subq      $24, %rsp

Body:
        movq      %rbx, vreg1
        movl      (vreg1), %eax
        subq      %rax, %rsp
        andq      $-16, %rsp
        movq      %rsp, %rax

x86_64 Epilogue:
        movl      %rbp, %rsp
        popl      %rbp
        movl      %rbx, %rsp
        popl      %rbx
        ret

m128 and PIC
int g_i;
void foo()
{
    volatile __m128 m = _mm_set_ps1(0.f);
    g_i = 123;
    ...
}

i686 Prologue:
        pushl     %ebp
        movl      %esp, %ebp
        andl      $-16, %esp
        pushl     %ebx
        subl      $16, %esp
        call      .L1
.L1:
        popl      %ebx
        ...

i686 Epilogue:
        addl      $16, %esp
        popl      %ebx
        movl      %ebp, %esp
        popl      %ebp
        ret

m128 + alloca + PIC
void foo(int size)
{
    char * ptr=alloca(size);
    volatile __m128 m = _mm_set_ps1(0.f);
    ...
}
i686 Prologue:
        pushl     %edi
        leall     8(%esp), %edi
        andl      $-16, %esp
        pushl     4(%edi)
        pushl     %ebp
        movl      %esp, %ebp
        subl      $24,  %esp
        call      .L1
.L1:
        popl      %ebx

Body:
        movl      %edi, vreg1
        movl      (vreg1), %eax
        subl      %eax, %esp
        andl      $-16, %esp
        movl      %esp, %eax

i686 Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        leal      -8(%edi), %esp
        popl      %edi
        ret

m128 + alloca + PIC + library call
void foo(int size)
{
    char * ptr=alloca(size);
    volatile __m128 m = _mm_set_ps1(0.f);
    printf("Hello\n");
    ...
}

i686 Prologue:
        pushl     %edi
        leal      8(%esp), %edi
        andl      $-16, %esp
        pushl     4(%edi)
        pushl     %ebp
        movl      %esp, %ebp
        subl      $24,  %esp
        call      .L1
.L1:
        popl      %ebx

i686 Body:
        movl      %edi, vreg1
        movl      (vreg1), %eax
        subl      %eax, %esp
        andl      $-16, %esp
        movl      %esp, %eax

Body:
        call      printf@PLT

i686 Epilogue:
        movl      %ebp, %esp
        popl      %ebp
        leal      -8(%edi), %esp
        popl      %edi
        ret

m128 and nested function and PIC
void foo()
{
    void bar(int arg1, int arg 2)
    {
         volatile __m128 m = _mm_set_ps1(0.f);
         ...
    }
    bar(1,2);
}

i686:
foo:
        ...
        movl      %ebp, %ecx
        call      bar@PLT
        ...
```

```
bar:
        pushl     %edi
        leal      8(%esp), %edi
        andl      $-16, %esp
        pushl     4(%edi)
        pushl     %ebp
        movl      %esp, %ebp
        subl      $24,  %esp
        call      .L1
.L1:
        popl      %ebx

        movl      %edi, vreg1
        movl      (vreg1), %eax
        subl      %eax, %esp
        andl      $-16, %esp
        movl      %esp, %eax
        ...

        movl      %ebp, %esp
        popl      %ebp
        leal      -8(%edi), %esp
        popl      %edi
        ret
```

m128, dynamic stack alloc and register parameter function call

```
static void bar(int arg1, int arg 2, int arg3)
{
    char * ptr=alloca(size);
    volatile __m128 m = _mm_set_ps1(0.f);
    ...
}

void foo()
{
    bar(1,2,3);
}

i686 foo:
        movl      $1, %eax
        movl      $2, %edx
        movl      $3, %ecx
        call      bar
        ...
bar:
        pushl     %edi
        leal      8(%esp), %edi
        andl      $-16, %esp
        pushl     $4(%edi)
        pushl     %ebp
        movl      %esp, %ebp
        subl      $24,  %esp

        movl      %edi, vreg1
        movl      (vreg1), %eax
        subl      %eax, %esp
        andl      $-16, %esp
        movl      %esp, %eax
        ...

        movl      %ebp, %esp
        popl      %ebp
        leal      -8(%edi), %esp
        popl      %edi
        ret
```

Thanks - Joey

- **Follow-Ups**:
    - **Re: A proposal to align GCC stack**
        - *From:* Christian SchÃler

Index Nav: [Date Index] [Subject Index] [Author Index] [Thread Index]
Message Nav: [Date Prev] [Date Next] [Thread Prev] [Thread Next]