GCC中内嵌函数实现剖析

李春江 杜云飞 易会战 杨灿群
(国防科学技术大学计算机学院 长沙 410073)

摘 要 GNU编译器集合(GCC)具有支持多种高级语言和多种目标处理器平台、文档及源代码开放等的特点,在工业界和学术界被广泛使用。GCC支持非常多的内嵌函数,内嵌函数是GCC编译器中非常重要的一部分实现。首先分析GCC中多种内嵌函数的目的和作用;之后结合实际工作,以使用向量扩展指令的内嵌函数实现为例,剖析了平台相关的内嵌函数的实现过程。本工作对深入理解GCC编译器中的内嵌函数实现机制,对基于GCC的研究和开发有较强的参考意义。

关键词 GCC,内嵌函数,剖析
中图法分类号 TP314 **文献标识码** A

Anatomy of the Implementation of Builtin Functions in GCC

LI Chun-jiang DU Yun-fei YI Hui-zhan YANG Can-qun
(School of Computer Science, National University of Defense Technology, Changsha 410073, China)

Abstract GNU Compiler Collection(GCC) supports multiple high level languages and multiple platforms with its internal documents and source code opened. It is widely used both in industry and research. GCC supports lots of builtin functions and the implementations of builtin functions are one of the important implementations of GCC. We analyzed the intensions and operations of the builtin functions in GCC. Then, combining with our practical work and taking the vector extension instruction oriented builtin functions as examples, we demonstated the implementation details of platform-related builtin functions in GCC. This work is really worth referenced both for the comprehension of the implementation mechanism of builtin functions and for research and development based on GCC.

Keywords GCC, Builtin functions, Anatomy

1 引言

GCC(Gnu Compiler Collection, Gnu编译器集合(简称GCC或GCC编译器))[1]是广泛使用的开源编译器套件。基于GCC编译器,编译优化的研究工作和编译器的实现工作非常活跃。

在编译器理论中,固有函数(Intrinsics)是指可以在特定高级语言中使用但由编译器特殊处理过的函数。在GCC编译器中,实现了多种内嵌函数(Builtin functions)作为固有函数;因此对GCC而言,Intrinsics和Builtin函数指向相同。GCC中针对C语言实现了几类内嵌函数,以支持在C语言程序中使用,其他高级语言也可以通过混合编程方法来使用这些内嵌函数。

本文首先简要介绍了GCC中几类内嵌函数面向的目标和作用;然后以实际工作中实现的支持向量扩展指令的内嵌函数为例,较详细地剖析了此类内嵌函数的实现细节。本文的工作对基于GCC的编译器研究开发有较大的参考价值。

2 GCC中的内嵌函数

GCC中包括以下几类内嵌函数:

1)存储器原子访问的内嵌函数;
2)使用向量扩展指令的内嵌函数;
3)象大小检查的内嵌函数;
4)平台相关的内嵌函数;
5)用内嵌汇编实现的内嵌函数;
6)其他内嵌函数。

2.1 存储器原子访问的内嵌函数

此类内嵌函数与GCC中其他类型的内嵌函数以"__builtin"作为函数名的开头不同,都是以"__sync"作为函数名的开头。并非所有的处理器平台都支持此类存储器原子访问的内嵌函数,具体处理器平台支持哪些函数需要进一步查阅相关手册。

存储器原子访问的内嵌函数大多数情况下都可以看作是"完全的存储器同步栅栏"。即,在此类函数调用点处,存储访

本文受国家自然科学基金项目多核多线程处理器SIMD扩展的编程模型和编译优化关键技术研究(61170046)和国家自然科学基金项目(61170045)资助。

李春江(1974-),男,博士,副研究员,主要研究方向为计算机体系结构、编译及优化技术,E-mail:chunjiangli@gmail.com;杜云飞(1980-),男,博士,助理研究员,主要研究方向为编译技术、系统软件;易会战(1976-),男,博士,副研究员,主要研究方向为计算机体系结构、编译及优化技术;杨灿群(1968-),男,博士,研究员,硕士生导师,主要研究方向为编译技术、系统软件。

问操作不能跨该函数调用点移动到之前或之后。而且，为了做到这一点，甚至需要编译器生成所需的指令来防止程序跨越此函数调用点进行前瞻的“取数操作”或者在此操作后将“存数操作”排队。简言之，“取数操作”和“存数操作”都不能跨越此函数调用点。

GCC 中此类存储器原子访问的内嵌函数与 Intel 公司的“Intel Itanium 处理器相关的应用二进制接口”文档[2]的 7.4 节兼容。

可见，存储器原子访问的内嵌函数实现了在 C 语言中对平台支持的原子操作的函数级编程接口。这种存储器原子访问的内嵌函数为用户编程进行存储器同步操作提供了高级语言编程支持。

表 1　GCC 中常用的存储器原子访问内嵌函数的形式和操作语义

存储器原子访问内嵌函数的形式	操作语义或解释
type __sync_fetch_and_add(type * ptr,type value,…) type __sync_fetch_and_sub(type * ptr,type value,…) type __sync_fetch_and_or(type * ptr,type value,…) type __sync_fetch_and_and(type * ptr,type value,…) type __sync_fetch_and_xor(type * ptr,type value,…) type __sync_fetch_and_nand(type * ptr,type value,…)	{ tmp= * ptr; * ptr op=value;return tmp;} { tmp= * ptr; * ptr=~(tmp & value);return tmp;} // nand 解释： 在对存储器中的内容操作前取出该值，在存储器中的内容修改后，仍返回存储器中原来的值。
type __sync_add_and_fetch(type * ptr,type value,…) type __sync_sub_and_fetch(type * ptr,type value,…) type __sync_or_and_fetch(type * ptr,type value,…) type __sync_and_and_fetch(type * ptr,type value,…) type __sync_xor_and_fetch(type * ptr,type value,…) type __sync_nand_and_fetch(type * ptr,type value,…)	{ * ptr=~(* ptr & value);return * ptr;} // nand 解释： 对存储器中的内容操作后返回修改后的值。
bool __sync_bool_compare_and_swap(type * ptr,type oldval type newval,…) type __sync_val_compare_and_swap(type * ptr,type oldval type newval,…)	解释：这些内嵌函数进行原子的比较并交换操作。
__sync_synchronize(…)	解释：存储器栅栏同步。
type __sync_lock_test_and_set(type * ptr,type value,…)	解释： 按 Intel 公司文档给出的描述，此内嵌函数并不是传统的测试并设置操作，而是一个原子的交换操作，把值写入 * ptr，返回 * ptr 原来的值。许多平台仅对这种锁操作提供最小的支持，并不支持全交换操作，在这种情况下，平台可能仅支持精简的功能即可写入的有效值只能是立即数 1，而 * ptr 中存储的实际的值是由平台实现定义的。这个内嵌函数并不是一个“全栅栏”同步，而是一个“获得栅栏(acquire barrier)”，这意味着在此内嵌函数后的存储器访问不能移动到函数调用点之前，但是函数调用点以前的存操作还未全局可见或之前的存储器取操作也没有完成。
void __sync_lock_release(type * ptr,…)	解释： 该内嵌函数释放__sync_lock_test_and_set 函数获得的锁，通常意味着向 * ptr 写入常数 0。这个内嵌函数也不是一个完全的“栅栏同步”，而是一个“释放栅栏”，这意味着前面所有的存储器存数操作是全局可见的，并且所有前面的存储器取数操作已经被满足，但后续的存储器读是可以前瞻性地调度到该栅栏之前。

2.2　使用向量扩展指令的内嵌函数

对于指令集中提供 SIMD(单指令多数据)向量指令的平台，GCC 在 C 扩展中实现了可直接映射为一条或多条 SIMD 指令的内嵌函数，可供 C 语言编程使用。当前主流的微处理器都实现了 SIMD 扩展，因此 GCC 面向不同处理器平台的向量扩展实现了非常多的内嵌函数。

2.3　对象大小检查的内嵌函数

GCC 通过实现对象大小检查的内嵌函数，支持有限的缓冲区溢出检查，来防范缓冲区溢出。函数原型如下：

*size_t __builtin_object_size (void * ptr,int type);*

其中 *type* 可取的值为“0,1,2,3”，即其最后 4 位取值情况影响了该函数的返回值。该函数返回从地址 *ptr* 开始到 *ptr* 所指向对象的结束位置的字节数目，条件是必须是编译时可知。这个内嵌函数不考虑其参数的副作用，如果参数存在副作用，那么该函数返回“(size_t)－1”(当 *type* 为 0 或 1 时)或返回“(size_t) 0”(当 *type* 为 2 或 3 时)。如果 *type* 的最低位为 0，那么对象是整个变量；如果其最低位为 1，那么对象是 *ptr* 所指向的最近的子对象。参数 *type* 的倒数第二位决定是返回最大还是最小的变量字节数。

还有几个类似的实现对象大小检查的内嵌函数，如用于检查存储器拷贝操作所拷贝对象大小的内嵌函数，以及用于对格式化输出函数进行缓冲区大小检查的函数；前者如__builtin__ memcpy_chk，后者如__builtin__sprintf_chk 等。具体使用时，须参考 GCC 文档中的相关内容。

这些对象大小检查的内嵌函数为用户编程提供了一些保护和检查的机制，合理使用这些内嵌函数可以增强软件的可靠性。

2.4　平台相关的内嵌函数

GCC 面向多种处理器平台，分别提供了特定平台相关的内嵌函数；这些内嵌函数用于支持特定平台指令集中的扩展指令。这些扩展指令中相当大的一类指令是支持 SIMD 计算的指令，如 x86 系列处理器中的 MMX[3]、SSE[4]、AVX[5]，PowerPC 的 AltiVec/VSX[6]，以及 SPARC 的 VIS[7]等。

2.5　用内嵌汇编实现的内嵌函数

GCC 支持用 asm(或__asm__)关键字来实现在 C 语言程序中嵌入汇编指令。在使用 asm 实现嵌入汇编的程序语句中可以用 C 表达式来作为汇编指令的操作数，这可以避免用户指定汇编指令使用的寄存器或存储器位置，而由编译器在寄存器分配过程中统一进行寄存器分配。

如下汇编语句取自 Linux 内核，其中 lo 和 address 都是 C

程序中的表达式，“=r”和“r”表示对操作数的约束，约束方式与 GCC 机器描述中使用的约束相同。在 GCC 中使用内嵌汇编的形式有多种[8]。

```
asm("ldlo.q %1,0,%0":"=r"(lo):"r"(address))
```

在 GCC 支持内嵌汇编语言的基础上，将完成特定功能的一条或多条内嵌汇编语句封装成函数，这为实现 GCC 内嵌函数提供了一种实现机制。这种实现机制适合于对处理器中特殊汇编指令（如控制处理器状态和模式的汇编指令）的内嵌函数支持。

2.6 其它内嵌函数

除了以上几类内嵌函数，GCC 还实现了一些编译器内部使用的内嵌函数，用于异常处理或变长参数列表处理，不建议用户使用这些内嵌函数，因为这些编译器内部使用的内嵌函数会随 GCC 的发展而改变。

GCC 中还实现了另外一些内嵌函数，可以分为如下两类：

• 标准 C 库函数的内嵌函数实现

对标准 C 库函数中的许多函数，GCC 都实现了一个内嵌函数版本，以“__builtin_”开头。许多这种实现的函数仅在某些情况下进行了优化，如果没有优化，那么对该内嵌函数的调用会定向到对标准 C 语言库函数的调用。

ISO C 模式下，许多函数如_exit，alloca，strndup，toascii 等，都有一个对应的以“__builtin_”开头的内嵌函数版本，这些函数甚至可以用在严格的 C90 模式下。除了在严格的 C90 模式下，ISO C94 中的函数如 iswalnum，iswalpha 等，以及 ISO C99 中的许多库函数如_Exit，acoshf，vsnprintf，vsscanf 等等，也被作为内嵌函数处理；除非编译的时候用-fno-builtin 编译选项来控制不使用内嵌函数。

• 其他内嵌函数

除了语言相关的内嵌函数外，GCC 还实现了多个内嵌函数，用于多种多样的目的。例如内嵌函数 void __builtin_trap (void)，其作用是使程序非正常退出。关于这些内嵌函数的详细情况请参见 GCC 在线文档[9]。

3 使用向量扩展指令的内嵌函数实现

在 OpenSparc T2 处理器内核[10]的基础上，扩展了支持四路双精度 SIMD 操作的向量处理单元（VPU）并设计了相应的指令集。在 GCC 的 Sparc 后端中实现了面向此类向量扩展指令的内嵌函数。这里以实际实现的指令为例，详细介绍此类内嵌函数的定义和实现过程。

3.1 内嵌函数初始化

在 sparc.c 中定义了初始化平台相关内嵌函数的宏：

```
#define TARGET_INIT_BUILTINS sparc_init_builtins
```

该宏是一个目标平台钩子（Target Hook）；在 target-def.h 文件的 TARGET_INITIALIZER 数组中包含了该宏；即在目标平台的各项初始化过程中，其中一项就是初始化平台相关的内嵌函数。

sparc_init_builtins 函数是初始化平台相关内嵌函数的具体实现，它调用 sparc_vis_init_builtins 函数；后者为支持 Sparc 的 VIS 指令，实现了相应的内嵌函数的定义和初始化工作。为了与设计实现的 VPU 单元的四路双精度 SIMD 指令对应，在此文件中设计了由 sparc_init_builtins 函数调用的 sparc_vpu_init_builtins 函数，在该函数中完成四路双精度 SIMD 指令的内嵌函数的定义和初始化。

3.2 内嵌函数定义

在 sparc.c 文件中定义使用向量扩展指令的内嵌函数。

为简化内嵌函数的定义，GCC 定义了如下的宏：

```
#define def_builtin(NAME,CODE,TYPE) \
add_builtin_function((NAME),(TYPE),(CODE),\
BUILT_IN_MD,NULL,NULL_TREE)
```

使用该宏就可以定义内嵌函数。其中 NAME 为函数名，CODE 是该函数对应机器指令的指令代码，TYPE 为函数的树类型。

例如，四路双精度向量加指令，在 sparc_vpu_init_builtins 函数中实现了如下定义：

```
def_builtin("__builtin_vpu_fvadd", \
CODE_FOR_vpu_fvadd,v4df_ftype_v4df_v4df);
```

其中的 CODE_FOR_vpu_fvadd 为指令代码。在编译 GCC 编译器的过程中，由 gencodes 函数解析机器描述文件（sparc.md）中的所有指令描述，生成 insn-codes.h 文件，该文件中为指令的指令代码赋值。编译过程器工作过程中，所有对相应指令的使用都可以通过该指令代码来索引。

v4df_ftype_v4df_v4df 是编译器后端中为实现内嵌函数而构造的在树一级中间表示函数的类型。定义该类型，需要如下几步：

a）定义 v4df 树结点类型

```
tree v4df=build_vector_type(double_type_node,4);
```

表示 v4df 树结点由 4 个双精度类型的树结点构成。

b）定义函数的树类型

```
tree v4df_ftype_v4df_v4df=build_function_type_list
(v4df,v4df,v4df,0);
```

其中 ftype 表示此树结点代表了一个函数；前面的 v4df 表示该函数返回值的树结点类型；后面的两个 v4df 表示该函数有两个参数，各个参数的结点类型都是 v4df。

4 Builtin 编程举例及编译器处理过程

```
#include <stdio.h>
typedef double V256 __attribute__((vector_size(32)));
#define N 1024
double src1[N] __attribute__ ((aligned (32)));
double src2[N] __attribute__ ((aligned (32)));
double res[N] __attribute__ ((aligned (32)));
int main ()
{
 int i;
 V256 res1, res2, res3, res4;
/* Initialization of source vector. */
 for(i = 0; i < N; i++)
 {
  src1[i] = (double)i * 0.1;
  src2[i] = (double)i * 0.1;
 }
/* Vector add using builtin functions. */
 for(i = 0; i < N/4; i++)
 {
  res1 = __builtin_vpu_fvload (&src1[i * 4]);
  res2 = __builtin_vpu_fvload (&src2[i * 4]);
  res3 = __builtin_vpu_fvadd (res1, res2);
  __builtin_vpu_fvstore (&res[i * 4], res3);
 }
 return 0;
}
```

图 1 内嵌函数编程举例

（下转第 379 页）

且中间的时间开销无疑降低了效率。

除此之外，本文还改造了 framework 的图形界面部分，实现了一个图形化的“Hello，World!”界面，具体的图形如图 5 所示。

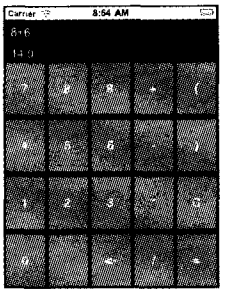

图 5　iOS 平台上的 Android 计算器

结束语　本文分析了 Dalvik 虚拟机的体系结构与运行机制，着重介绍了 iOS 平台的特点，同时给出了移植的可行性分析、内存管理模块的实现、解释器的重写以及 IPC 机制的改写这 4 个与平台相关的分析与策略，移植后虚拟机可以在 iOS 平台上提供 Java 运行环境。本文在撰写的同时，也进行了 Dalvik 底层相关类库的移植，旨在为 Android 的程序提供图形化界面、网络通信、数据库支持等多样化的服务，所以我们下一阶段的主要研究工作就是在不久的将来，让所有的 Android 的应用程序都能够在我们所移植的虚拟机上运行。

参 考 文 献

[1] Wiki，“Dalvik(software).”[EB/OL]. http://en.wikipedia.org/wiki/Dalvik(software)

[2] Google Inc，”Dalvik Porting Guide.” [EB/OL]. Available: http://android.git.kernel.org/

[3] 周毅敏，陈榕. Dalvik 虚拟机进程模型分析[J]. 计算机技术与发展，2010，20(2)：83-86

[4] 陈卫伍，王建民，陈榕. Dalvik 在 CAR 构件运行时中的应用研究[J]. 电脑知识与技术，2010，6(31)：8865-8868

[5] 吴少刚，邹国民. Dalvik 虚拟机在龙芯平台上的研究与移植[J]. 计算机工程，2011，37(16)：1-4

[6] 叶云，李春强，胡军山. 基于 CK610 的 Dalvik 虚拟机移植与优化[J]. 计算机工程，2011，37(16)：291-293

[7] 陈灏，陈榕. 支持复合对象的 Java 虚拟机内存管理技术研究[J]. 电脑知识与技术，2011，7(22)：5356-5359

[8] 苏超云，柴志雷，涂时亮. 实时 Java 平台的类预处理器研究[J]. 计算机工程，2010，36(7)：246-251

[9] 孟小华，区业祥. 一种 J2ME 应用向 Android 平台移植的方案[C]// Proceedings of 2010 International Conference on Future Information Technology and Management Engineering. 2010(3)：60-63

(上接第 359 页)

本文设计实现的面向四路双精度短向量 SIMD 指令的内嵌函数，为面向 SIMD 的编程提供了高级语言级的 API，极大地方便了面向 VPU 部件的程序开发。图 1 给出了用内嵌函数编写的一维向量加程序。

在编译器对该程序的编译过程中，在树一级的程序变换过程中内嵌函数保持原始状态，在树一级最后的优化遍 140t.optimized 之后，如上程序中内嵌函数的形式，如图 2 所示。当进入 RTL[9]（寄存器传输语言，GCC 编译器机器无关的后端所使用的中间表示）编译阶段的第一遍 141r.expand 时，这些内嵌函数转换为相应向量指令的 RTL 形式，如图 3(a)所示。在转换到 RTL 代码之后，经过相应的优化遍和寄存器分配输出的汇编代码形式如图 3(b)所示。

```
D.2009_13 = &src1[ivtmp.17_22];
res1_14 = __builtin_vpu_fvload (D.2009_13);
D.2010_18 = &src2[ivtmp.17_22];
res2_19 = __builtin_vpu_fvload (D.2010_18);
res3_20 = __builtin_vpu_fvadd (res1_14, res2_19);
D.2011_24 = &res[ivtmp.17_22];
__builtin_vpu_fvstore (D.2011_24, res3_20);
```

图 2　树一级优化后的内嵌函数形式

```
(insn 87 86 88 6 example.c:20 (set (reg:V4DF 218)
    (mem:V4DF (reg:DI 223) [0 S32 A256])) -1 (nil))

(insn 88 87 89 6 example.c:20 (set (reg/v:V4DF 198 [ res2 ])
    (reg:V4DF 218)) -1 (nil))

(insn 89 88 90 6 example.c:21 (set (reg/v:V4DF 199 [ res3 ])
    (plus:V4DF (reg/v:V4DF 195 [ res1 ])
        (reg/v:V4DF 198 [ res2 ]))) -1 (nil))
```

(a)

```
ldv    [%g3+%g2], %v1
ldv    [%g4+%g2], %v0
fvaddd %v1, %v0, %v0
```

(b)

图 3　对应指令的 RTL 形式及汇编语言输出形式

结束语　内嵌函数在 GCC 编译器中扮演重要的角色，恰当使用 GCC 中的内嵌函数进行软件开发有利于面向体系结构平台编写高性能的软件或者为软件提供更高的可靠性保障。本文首先分门别类分析了 GCC 中多种内嵌函数的目的和作用；之后结合实际工作，以使用向量扩展指令的内嵌函数实现为例，剖析了平台相关的内嵌函数的实现过程。本文的工作对深入理解 GCC 编译器中的内嵌函数实现机制，对基于 GCC 的研究和开发有很强的参考意义。

参 考 文 献

[1] [OL]. http://gcc.gnu.org/

[2] Intel ItaniumTM Processor-Specific Application Binary Interface (ABI). Intel Corporation，May 2001

[3] Intel Architecture Software Developer's Manual Volume 2：Instruction Set Reference. Intel Corporation，1999

[4] Intel. SSE4 Programming Reference. Intel Corporation，2009

[5] Firasta N，Buxton M，Jinbo P，et al. Intel AVX：New Frontiers in Performance Improvements and Energy Efficiency[J]. Intel white paper，2008

[6] Eisen L，Ward J W，et al. IBM POWER6 accelerators：VMX and DFU，IBM Corporation，2007

[7] The VISTM Instruction Set V1.0. White paper，Sun Microsystems Inc.，June 2002

[8] Coleman C L. Using Inline Assembly with GCC. January 2000

[9] GCC Internals[EB/OL]. http://gcc.gnu.org/onlinedocs/gcc-4.6.1/gccint/index.html

[10] OpenSPARCTM T2 Core Microarchitecture Specification，Revision A. Sun Microsystems，Inc.，December 2007

[11] Stallman R M. Tthe GCC Developer Community. GNU Compiler Collection Internals. For GCC version 4.6. Free Software Foundation. 2010