Dalvik在iOS上的移植

高士翔[1] 葛 霄[1] 潘 磊[2] 郑 滔[1,2]
(南京大学软件学院 南京 210093)[1] (南京大学计算机软件新技术国家重点实验室 南京 210093)[2]

摘 要 在分析Dalvik虚拟机体系结构设计的基础上,针对iOS平台研究了Dalvik移植过程中的几个关键技术,并在iOS平台上成功构建了基于Dlavik的JAVA运行环境,这对iOS手机与Android手机的跨平台应用开发具有重大应用价值。对移植后的Dalvik进行了性能分析,给出了结论,并给出了下一步的项目计划。

关键词 Android,iOS,Dalvik,JAVA虚拟机,可移植性

Portability of Dalvik in iOS

GAO Shi-xiang[1] GE Xiao[1] PAN Lei[2] ZHENG Tao[1,2]
(National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)[1]
(Software Institute, Nanjing University, Nanjing 210093, China)[2]

Abstract We analyzed the Dalvik virtual machine architecture, and studied several key technologies about the process of porting Dalvik for the iOS platform. We successfully built a Dlavik based JAVA runtime environment on the iOS platform which has great application value in mobile application development of the iOS and Android cross-platform. We also analyzed the performance of the Migrated Dalvik and gave a conclusion and the fulture plan of this project.

Keywords Android, iOS, Dalvik, Java virtual machine, Portability

随着移动技术的发展以及手机的普及,大屏幕触控智能手机越来越受到人们的青睐,尤其以iPhone为代表的新一代手机正在改变手机市场格局。苹果公司自2007年首发iPhone以来,经历五代的发展,凭借其独特的设计理念与运营方式,成为手机行业的佼佼者。据调查显示,2011年上半年iPhone在全球智能手机市场的份额达到16.01%。报告同时称,由于iPhone进入中国市场,并且在多个国家增加了多个运营商,iPhone的市场份额还将继续上升。与此同时,谷歌公司推出的Android平台及其相关的手机占据了市场的一定份额,具有很高的发展潜力。iOS和Android平台已经成为手机应用开发者选择最多的两个平台。对移动平台开发者的调查显示,20%的人同时为iPhone和Android编程。但是,两个平台存在着巨大的差异性,开发者为了同一款应用进行两次开发,导致不必要的成本开销。如果通过技术手段来实现两个平台的融合,势必可以为开发者节约成本,缩短开发周期,产生积极而深远的影响。

Dalvik是谷歌公司设计用于Android平台的Java虚拟机。它支持已转化为.dex格式的Java应用程序。在众多JVM版本中,Dalvik更能适应内存大小和处理器速度有限的系统。因此,本文选定Dalvik为移植对象,在iOS平台上构筑了Java运行环境[1]。

目前国内外已经对Dalvik的体系结构以及在不同平台上的移植工作等进行了一系列的深入研究,如周毅敏等分析了Dalvik的进程管理模块并分析了进程模型[3];陈卫伍等构想了在Dalvik平台上,利用Java和CAR混合编程技术来提高JAVA运行效率的方法[4];吴少刚等进行了Dalvik在龙芯平台的移植与优化工作[5];叶云等着重分析了Dalvik的解释器、本地方法桥以及C库在CK610平台的移植与优化[6];孟小华等针对J2ME应用分析了Android与J2ME的异同,并给出了一种高效可行的J2ME移植方案[9]。

本文第1节分析和研究了Android平台的Dalvik的体系结构,简要地介绍了Dalvik虚拟机中各模块的特性以及整体的运行机制,点明了Dalvik与其他虚拟机的不同之处;第2节叙述了iOS平台的相关开发规范与限制,阐明了iOS平台特性对本文移植工作的影响;第3节提出了移植的解决方案;第4节给出了移植工作的实验设计与结果;最后是对本文工作的总结以及对未来工作的展望。

1 Dalvik虚拟机分析

1.1 Dalvik介绍

虚拟机是通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。

虚拟机按工作性质和功能特性可分为以下两种:

1)系统级虚拟机:支持整个操作系统的运行。

2)进程级虚拟机:支持单个进程的运行。

按其体系结构可分为以下两种:

1)基于栈式虚拟机:指令装载入栈中实现;

2)基于寄存器式虚拟机:指令装载入寄存器中实现。

Dalvik是基于寄存器式进程级虚拟机。手机内存低,CPU慢,电池容量小。相较于其他虚拟机,Dalvik在这种环

本文受国家自然科学基金(60875038,61105069),江苏省科技支撑计划(BE2009142,BE2010142)资助。
高士翔(1991-),男,本科生,主要研究方向为虚拟机移植;葛 霄 本科生;潘 磊 博士生;郑 滔 教授。

境下表现更优异，并且 Dalvik 的字节码验证机制在一定程度上防止了恶意代码的执行[2]。

1.2 Dalvik 虚拟机体系结构设计

Dalvik 虚拟机的体系结构如图 1 所示，主要有 5 个模块。

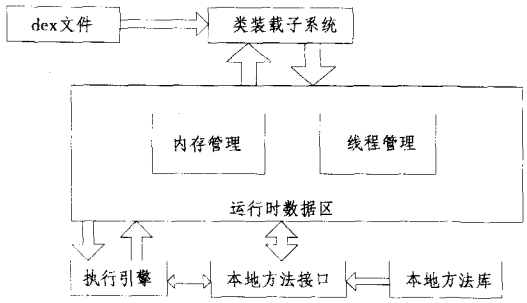

图 1 Dalvik 虚拟机体系结构

(1)类装载模块：用于抽象装载类的过程，与底层的数据存储无关。(2)内存管理模块：核心分为两个部分：内存分配和内存回收。Java 使用 new 操作符进行内存分配，但是与 C/C++等语言不同的是，Java 并没有提供任何操作来释放内存，而是通过垃圾收集机制来进行内存回收。内存回收的技术核心为垃圾回收机制。(3)线程管理：在 Java 多线程支持的基础上，该模块主要处理线程调度、线程同步以及栈内存的管理。(4)执行引擎：负责解释执行 dex 字节码。(5)本地方法接口：提供 Java 语言调用 C/C++。

1.3 Dalvik 和 Java VM 的比较

1.3.1 内存开销

在内存消耗方面，Java VM 和手机系统存在一对尖锐的矛盾。前者使用大量的内存资源来支持它的垃圾回收机制，后者则由于运行速率的限制，需要大量节约内存[7,8]。Dalvik 设计了.dex 文件格式，极大地减少了虚拟机运行时的内存需求。同时 Dalvik 基于 Zygote 的孵化器实现了进程间的代码共享，同样减少了内存的开销。

1.3.2 体系结构

基于寄存器的 Dalvik 既提高了 CPU 的运行效率，又为大型计算提供了更多并行计算的机会。

1.3.3 JIT 引擎

每启动一个应用程序，都会相应地启动一个 Dalvik 虚拟机，启动时会建立 JIT 线程，其一直在后台运行。当某段代码被调用时，虚拟机会判断它是否需要编译成机器码，如果需要，虚拟机会为其进行标记，JIT 线程不断判断此标记，如果发现被设定就把它编译成机器码，并将其机器码地址及相关信息放入 entry table 中，下次执行到此就跳转至机器码段执行，而不再解释执行，从而提高速度。

1.3.4 可靠度

在标准的 Java 运行环境中，一个模块的异常会引起系统的崩溃，而每个独立进程在 Dalvik 中都拥有独立空间，单个应用程序的异常不会导致整个系统的崩溃。

2 iOS 平台特点及编程规范

2.1 iOS 平台提供的编程框架

iOS 是 iPhone、iPad、iPod Touch 设备的操作系统。其系统结构自底向上可分为：核心操作系统层、核心服务层、媒体层、可轻触层。

核心操作系统层提供了整个 iOS 的一些基础功能，比如：硬件驱动、内存管理、程序管理、线程管理、文件系统、网络，以及标准输入输出等等，这些服务都可由 C 语言程序使用。

核心服务层在系统层之上提供了更为丰富的功能，它包含 Foundation. Framework 和 Core Foundation. Framework。

具体提供了字符处理、时间日历、数据安全、SQLLite、电话本等功能。Foundation 可供 Objective-C 语言使用，Core Fundation 可供 C/C++语言使用。

再往上是媒体层和可轻触层。媒体层主要提供图片、音乐、影片等功能，而可轻触层则提供应用程序的组件架构，以及处理屏幕上的触摸、文字输出等功能。大部分的 iOS 应用程序是利用媒体层与可轻触层进行开发的。

2.2 iOS 沙盒限制

为安全起见，iOS 将所有 SDK 开发限制在应用程序“沙盒”中(引用)。在使用苹果公司提供的 IDE 开发 iOS 应用程序时，IDE 会生成一个文件夹以及一个 ID 来标识该程序，这就是所谓的程序“沙盒”。在“沙盒”中，有 3 条原则必须毫无保留地遵守：

(1)应用程序只能在自己的“沙盒”中运行，其他“沙盒”对其透明。

(2)数据私有化。应用程序所使用的文件属于“沙盒”独有，这意味着“沙盒”间没有文件访问和移动权限。

(3)应用程序拥有自己的 Library、Documents 和/tmp 文件夹。它们由 IDE 自动生成，归“沙盒”独有，起到对数据访问与写入的限制作用

除了这些限制之外，应用程序必须具有数字签名，通过编码的应用程序标识符向操作系统证明自己的身份，该标识符必须在苹果公司的开发人员计划网站创建(引用)。一方面，这保证了无论何时 iOS 设备插入电脑，应用程序的数据都可及时更新。但是另一方面不利的是，这种设计限制了从 Windows 和 Mac 主机直接访问数据的能力。

2.3 其他限制

2.3.1 开发范围限制

iOS 设备由于体积大小的限制，不能提供与 PC 主机相当的磁盘空间。除此之外，开发人员必须时刻关注界面交互和电量消耗等问题。这些都对研发工作造成影响，开发人员必须在效果与性能两者间有所妥协。

具体而言，苹果公司规定：不能使用 iOS 设备针对大屏幕、鼠标、物理键盘，甚至物理常开 A/C 电源进行设计。所幸的是，苹果公司设计的 iOS 在众多手机平台中出类拔萃，它在存储、交互控制和电池寿命的局限下充分地展现了灵活性。

2.3.2 内存限制

在开发 iOS 应用程序时，必须小心地进行内存管理。iOS 并没有基于磁盘交换的虚拟内存机制，这意味着内存资源十分珍惜。当内存耗尽时，iOS 将会自动重启。因此，应该尽量以技术手段减少对内存资源的依赖，以防止 iOS 终止应用程序。开发者还需注意应用程序所使用的资源或文件内容。过多高分辨率的图像或编码率过高的音频文件会导致应用程序自动终止。

2.3.3 电量限制

电量限制一直是手机平台开发不可忽视的问题。苹果公司充分考虑了这一问题，以限制应用程序对 CPU 和电量的无

节制使用。同时开发者还应测试长时间使用下应用程序是否会导致手机过热，如果有这种情况，灵活地使用技术缓解这一情况，以防止对 iOS 设备硬件的过度损耗。

2.3.4 应用程序限制

苹果公司提出了一项强有力的“一次一个应用程序”策略（引用）。这意味着苹果官方不允许第三方开发人员开发在后台运行的应用程序。当用户使用一个应用程序时，iOS 必须停止当前程序，将当前程序状态信息保存，然后才将控制传递给用户选择的应用程序。另外，不能让检查新消息或定期发送更新的后台程序一直运行下去。

另一方面，苹果公司要求应用程序的推送服务必须由用户主动许可，后方才能开启。允许用户可以在 iOS 设备的设置中关闭这些推送服务。

如果在开发过程中违反上述的限制以及苹果其他相关的平台开发限制，无论是开发者自身导致的还是使用了违反规定的“Open SDK”，所开发出的应用程序都不允许在 App Store 上销售。

3 Dalvik 在 iOS 平台上的移植

考虑到 iOS 平台的特有性质，本文的移植工作将重点考虑内存管理和解释器这两大模块的移植，而平台间的差异性以及兼容性等问题将会成为移植工作的难点。另外，上面提到的应用程序限制不支持 Android 的 IPC 机制，即进程间通信机制。要实现 Android 程序在 iOS 上运行，必须改写 IPC 机制。

3.1 技术可行性分析

3.1.1 平台的语言兼容性

Dalvik 本身是由 C 语言编写的，这种特性决定着其移植的平台必须支持 C 语言的运行环境，而 iOS 平台具备这样的条件，所以移植过程中最大的障碍——语言障碍被克服了[2]。

3.1.2 POSIX 库支持

iOS 遵循了 POSIX 规范，在类库层次上支持了 Dalvik 的移植实现。

3.1.3 Dalvik 良好的模块设计

松耦合的设计理念使得 Dalvik 的各个模块均可以拆分、替换，并且 Dalvik 在代码中设计了大量的预处理机制来实现模块的加载控制，这对本文控制整个移植流程提供了极大的帮助。

3.2 内存管理模块的实现

Dalvik 内存管理模块主要移植部分是 dlmalloc。Dlmalloc 采用所谓的边界标记法将内存划分成很多块，从而对内存的分配与回收进行管理。本文为 dlmalloc 的实现源码定义了两种结构体 malloc_chunk 和 malloc_tree_chunk 来描述这些块，小于 256 字节的 chunk 块由结构体 malloc_chunk 来描述，大于 256 字节的 chunk 块由结构体 malloc_tree_chunk 来管理。

结构体 malloc_chunk 和 malloc_tree_chunk 的定义如下：

```
struct malloc_chunk {
    size_t prev_foot;
    size_t head;//Size and inuse bits
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
typedef struct malloc_chunk mchunk;
typedef struct malloc_chunk * mchunkptr;
typedef struct malloc_chunk * sbinptr;

struct malloc_tree_chunk {
    with malloc_chunk
    size_t prev_foot;
    size_t head;
    struct malloc_tree_chunk * fd;
    struct malloc_tree_chunk * bk;
    struct malloc_tree_chunk * child[2];
    struct malloc_tree_chunk * parent;
    bindex_t index;
};
```

结构体 malloc_chunk 与结构体 malloc_tree_chunk 管理方式相似，因此本文以 malloc_chunk 介绍内存管理方式，其栈空间如图 2 所示。

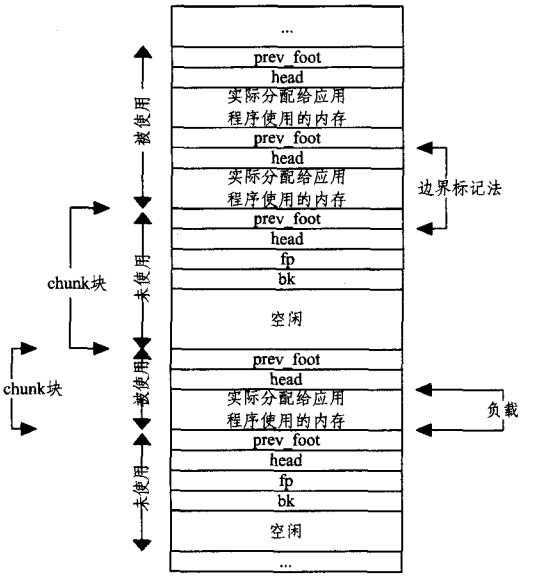

图 2 边界标记法划分内存

按照边界标记法，结构体 malloc_chunk 通过 prev_foot 和 head 将内存分割成很多块。字段 head 记录本 chunk 块大小、本块是否在使用中、前一 chunk 块是否在使用中等相关信息。

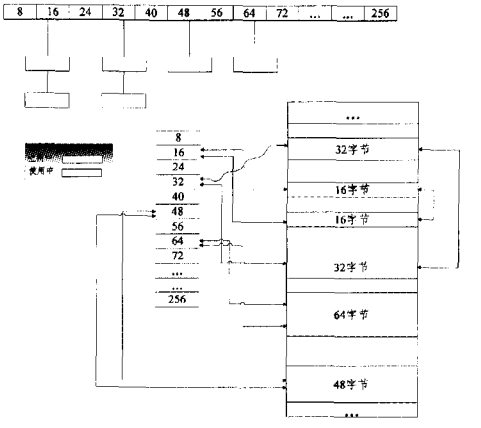

图 3 分箱式内存管理

head 字段的存储能力之大，其关键是 dlmalloc 在分割地址时总是按照地址对齐的方式进行，而块大小最小为 8 个字节并且设置的值必须是 2 为底的幂函数的值，所以 head 存储

块大小信息后末 3bit 总是 0，可以用来存储其他信息。

根据上面分析的 iOS 平台的内存限制，本文采用分箱式内存管理来减少内存开销。如图 3 表示 dlmalloc 对 256 字节以下的空闲 chunk 块的组织方式。

现有的方法是申请 32 个 malloc_chunk 结构体指针数组，再将其建立成具有头结点的链表。但是链表中 malloc_chunk 中的 prev_foot 和 head 便失去了意义，还会浪费内存空间。本文选择申请一个名为 smallbins 的具有 66 个 malloc_chunk 结构体指针元素的数组。这样所占空间为 66 * 4=264 字节，满足了 32 * 8=256 字节的内存需求。

3.3 解释器的移植

Dalvik 的解释器部分由汇编语言编写。由于汇编语言具有平台特异性，因此在移植过程中不能继续使用原来的解释版本。根据上面分析的平台语言兼容性，本文选择用 C 语言重写解释器部分。Dalvik 大约支持 248 种字节码指令，每个指令包含单字节的操作码和若干个操作数。

本文采用 switch/case 结构设计，来进行指令的识别与处理，实现虚拟机的解释器模块对于大部分的 cpu 计算指令，Dalvik 利用 Java 栈实现上层的抽象。其中与解释器模块结合最紧密的是操作数栈。一般是把使用的操作数值压入栈中。虽然 Dalvik 中没有保存任何数值的寄存器，但每个方法都有一个局部变量集合。本文的指令集实际的工作方式是把局部变量当作寄存器，用索引来访问。

本文定义了一个 32 位的指针结构 fp，即 frame point，以模拟栈指针在栈中的定位作用。对于一般的计算指令，解释器在完成该指令的执行之后会将 fp 向下移动 32 位以指向下一条指令。另外一些指令，例如 goto，returnAddress 这样的指令，解释器决定下一条指令时会把它当作当前执行指令的一部分。假如一条指令抛出异常，解释器将搜索合适的 catch 字句，以决定下一条指令。该指针初始存在于线程所申请的一块内存空间之中。当调用新的方法时，本文通过向上移动 fp 来开辟一块 SaveArea，并记录返回地址等信息，将 fp 指向新的位置，实现函数的递归调用。

3.4 IPC 机制的改写

IPC 机制作为 Android 系统中一种设计思想，以进程间的共享信息为基础，将应用级行为与系统级行为剥离，提高了系统的可扩展性与安全性。其架构如图 4 所示。

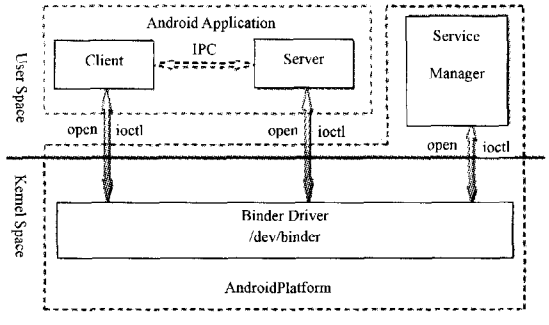

图 4 Android 进程间通信(IPC)机制架构

1. 4 个模块分处在不同的内存空间中。Client、Server 与 Service Manager 在用户空间中实现，Binder 驱动程序在内核空间中实现。

2. Binder 与 Service Manager 已由 Android 平台实现，开发者只需要在用户空间实现自己的 Client 和 Server。

3. Binder 驱动程序提供设备文件/dev/binder 与用户空间交互，Client、Server 和 Service Manager 通过 open 和 ioctl 文件操作函数与 Binder 驱动程序进行通信。

4. Client 和 Server 之间的进程间通信通过 Binder 驱动程序间接实现。

5. Service Manager 是一个守护进程，用来管理 Server，并向 Client 提供查询 Server 接口的能力。

针对 iOS 的多进程限制，设计了一套解决方案。原 Android 程序中，系统通过 IPC 机制将服务模块化，Client 模块向 Server 模块提出服务申请。将 4 个独立的模块整合到 Client 模块之中，通过函数调用的方式进行服务申请，改变了原先 C/S 的设计架构，将多进程转化为单进程，以满足上面提到的应用程序限制。通过改写 IPC 机制，本文将实现 Android Framework 的移植。

4 实验与结果分析

Instruments 是集成在 Xcode 中的一款性能测试软件，可以检测程序运行中的 CPU、内存占用等参数。

本文将移植后的 Dalvik 命名为“iDalvik”。由于每次运行 Android 程序都需要先在 iOS 上运行 iDalvik，因此 iDalvik 的性能至关重要。只有 iDalvik 完全启动之后，才能在 iOS 平台上运行后续的 Android 应用程序。

因此，本文利用上文介绍的工具，首先在 Xcode 模拟器上对 iDalvik 进行了性能测试。再在 Xcode 模拟器上运行 iDlavik，并记录其在 iOS 系统上所占用的各种资源，以及 iDalvik 完全启动所需要的时间。结果如表 1 所列。

表 1 iDalvik 性能测试

性能项目	运行时间/ms	内存/MB	使用堆栈/MB
iDalvik	898	72.1	58.19

为了验证 JAVA 运行环境已经在 iOS 平台上成功构建，本文选取了两个有代表性的测试案例进行测试。

首先，构造了一个测试用例。测试用例使用 JAVA 的 System. out. println()接口，测试 iOS 环境下是否具备 Java 环境，以及代码重复循环工作的能力。

其次，构造了第二个测试用例，该测试用例包含两个部分，第一部分是 1 个测试类，该类有 5 个基本数据类型组成；第二部分包含 main 函数，用于初始化测试类实例，并且重复 1 万次。该测试用例用于测试 iDalvik 是否具备在 iOS 环境下的内存管理能力。

具体如表 2 所列。

表 2 JAVA 运行环境构建测试

	运行时间/ms	使用堆栈/MB
循环输出一千次	837	0
创建一万个小对象(对象含五个成员变量)	56	0.53

经测试在 iOS 上两个测试用例都可以正常运行，说明运行环境构建成功。但是其运行时间较长，其原因大致有以下两点：

(1)由于移植需要，本文改写了 Dalvik 的解释器。原解释器由汇编语言编写，其运行效率高于本文所设计的 C 语言版本的解释器。

(2)Android 上的 Dalvik 是在 linux 内核上直接运行的，而移植后 Dalvik 在 iOS 架构的最上层作为应用程序运行，并

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。

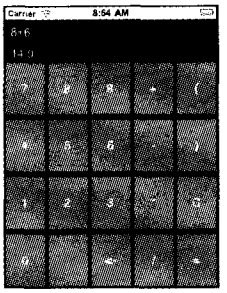

图 5　iOS 平台上的 Android 计算器

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