Journal of Computer Applications
计算机应用,2012,32(3):823 -826,834
ISSN 1001-9081
CODEN JYIIDU
2012-03-01
http://www.joca.cn

文章编号:1001 -9081(2012)03 -0823 -04
doi:10.3724/SP.J.1087.2012.00823

对用户交互响应进行加速的即时编译技术

刘　丽[1],古幼鹏[2*],唐德波[1]
(1.中国西南电子技术研究所,成都 610036;　2.中兴通讯股份有限公司 成都研究所,成都 610041)
(*通信作者电子邮箱 gu.youpeng@zte.com.cn)

摘　要:对于影响用户交互响应速度的瓶颈代码段,现有即时编译器存在无法准确选取和在程序启动阶段没有可用的本地码进行加速的问题,这影响了即时编译技术在用户交互响应方面的加速效果。为此,对即时编译器原有的代码选择策略和编译模式进行了改进。在代码选择策略方面,应用程序可以根据实际运行情况主动选择要编译的代码段,保证所有影响用户交互响应速度的瓶颈代码段都能被选取并被加速;在编译模式方面,本次编译得到的本地码可以保存并供程序下次运行时使用,保证在程序启动阶段也有本地码可用来加速。应用程序启动速度的实验表明,改进的即时编译器能够提升1倍的用户响应速度。

关键词:即时编译;嵌入式系统;用户交互;响应速度;编译模式;代码选择
中图分类号:TP312JA　　**文献标志码**:A

Just-in-time compilation for improving response speed of user interaction

LIU Li[1], GU You-peng[2*], TANG De-bo[1]
(1. *Southwest China Institute of Electronic Technology, Chengdu Sichuan* 610036, *China*;
2. *Institute of Chengdu, ZTE Corporation, Chengdu Sichuan* 610041, *China*)

Abstract: For the bottleneck code that impacts the user interaction speed, the current Just-In-Time (JIT) compiler cannot select it accurately or accelerate it during program start-up phase. The code selection strategy and compiling mode of current JIT compiler were improved in this paper. According to the new code selection strategy, application could select the code to be compiled on its own initiative in a given situation, which ensured all the bottleneck codes to be selected and accelerated. As for the new compiling mode, the native code could be saved and be used for the next program running, which ensured bottleneck code to be accelerated even during program start-up phase. The experimental result shows that the response speed of user interaction by using the improved JIT compiler is about two times that by using the old JIT compiler.

Key words: Just-In-Time (JIT) compilation; embedded system; user interaction; response speed; compiling mode; code selection

0　引言

Java具有跨平台、高安全性、开发容易的优点,被许多嵌入式系统选为编程语言。近来,随着移动操作系统Android[1]的成功,Java更成为了智能手机、平板电脑(tablet)等移动设备的主流编程语言。但是,Java具有性能上的缺陷。即时(Just-In-Time, JIT)编译技术是提高Java性能的有效手段,已经被当前大多数Java虚拟机广泛采用,并成为其中一项核心功能[2],其基本思想是:在字节码被第一次执行之前,先把字节码编译成本地码(即动态编译),然后再执行本地码。用本地码代替字节码来执行可以大幅度提高性能,但字节码的编译过程会消耗CPU和内存资源,又会对性能产生不利影响。嵌入式设备的CPU和内存资源非常有限,字节码编译过程对性能的不利影响会更加严重。因此,嵌入式JIT技术研究的主要问题是如何降低字节码编译的不利影响和提高编译质量。其中,文献[3-4]提出的部分编译字节码、解释与本地码混合执行模式的思想,利用了"软件运行过程中少量代码占据了大量执行时间"的统计结果,只花最小的代价(编译很少量的代码)就获得了最大的效果(在大部分执行时间里都能加速),是目前嵌入式JIT技术的基础;文献[5]则进一步表明:编译的字节码按照执行频率来选取的话,编译对CPU和内存资源的消耗几乎可以忽略,而程序总体性能的提高却非常有效。在此基础上,其他的研究工作致力于进一步提高编译过程的速度和编译质量。文献[6]改进了编译过程中的本地码优化阶段所使用的算法来提高编译速度;文献[7]把多个字节码组合成更粗粒度的superoperators,在superoperators层次上执行二进制代码优化,减少了编译过程中优化操作的执行次数和时间,同时代码优化可以更有针对性,提高了本地码的质量;文献[8]使用限制编译的字节码单位为方法来降低编译过程的复杂性,直接使用解释器的codelet作为本地码代码的生成模板,减少了本地码优化的工作量;文献[9]把编译过程中常用的静态单赋值(Static Single Assignment, SSA)方法改进成了TSSA(Trace SSA)方法,提高了编译的本地码质量;文献[10]采用一种"消极"的生成本地码的方式,在内存和CPU资源紧张的情况下生成高质量的本地码;文献[11]针对XScale处理器的体系结构特性优化本地码,让生成的本地码尽量利用处理器的硬件特性提高运行速度,提高编译质量。最后,文献[12]中的嵌入式JIT技术主要解决嵌入式系统中的另一个问题——实时性,即消除由于JIT编译引起的实时任务的执行截止时间不确定的问题。总之,现有的嵌入

收稿日期:2011-08-18;**修回日期**:2011-11-17。
作者简介:刘丽(1977-),女,四川遂宁人,助理工程师,主要研究方向:嵌入式系统、航空电子;　古幼鹏(1974-),男,重庆人,高级工程师,博士,主要研究方向:嵌入式系统、移动计算系统、移动互联网;　唐德波(1975-),男,四川广安人,工程师,硕士,主要研究方向:嵌入式系统、航空电子。

式 JIT 研究主要集中在性能和实时性方面。

与传统的嵌入式设备相比,移动设备以人为中心,用户交互的友好性(用户体验)是影响这类设备能否成功的重要因素。用户交互响应速度是用户体验的重要内容,虽然现有 JIT 研究关注的焦点问题——性能问题可以对提高用户交互响应速度起到很好的作用,但性能与用户响应速度并不能完全画等号[13]。随着移动设备的发展以及 Java 在这类设备中的广泛应用,对 JIT 提高用户交互响应速度进行专门研究具有积极意义。

1 用户交互响应加速的基本思想

用户交互响应速度是指系统对用户操作的响应速度。不同设备、不同输入,有不同的响应速度要求。因此,某个设备的用户交互响应速度是用户使用该设备时所有可能的输入的响应速度的集合。用户所有可能的输入可以分成 3 种:1)频繁的操作;2)偶发的操作;3)启动程序的操作。频繁和偶发的操作根据用户在程序正常运行时间段内使用该操作的频率来区分。例如,在通讯录程序正常运行时间段内,查找某个联系人是频繁的操作,新建联系人就是偶发的操作。启动程序的操作是一种特殊的用户输入操作,需要单独分析。与 PC 上的软件使用模式不同,移动设备上的软件使用模式具有程序启动频繁、程序运行时间短暂的特点。以用手机打电话为例,其软件使用模式为:启动通讯录程序、查找某个人的电话号码、退出通讯录程序、启动通话程序、输入查找到的电话号码、拨号并通话、退出通话程序。这个流程启动了两次程序,每次程序运行时间都很短暂。如果启动程序操作时间长,用户为使用每个(程序的)功能都要等待较长的程序启动时间,会使用户体验和设备的使用效率变差(很大一部分时间花费在等待上,无法使用设备)。然而程序启动操作由于要做很多运行前的准备工作,往往又非常耗时。因此,程序启动操作在整个系统的响应速度中占有特殊的地位,是一个基础性指标。

提高响应速度关键是提高与用户输入操作处理相关的瓶颈代码段的运行速度。这些代码段一般具有执行频率较高的特点,但是,也有一些瓶颈代码段的执行频率比较低。例如,在移动通信网络环境下,为了节省网络资源,往往采用按需建立数据链路的方法。当用户使用浏览器上网时,访问网页是这种场景下的频繁操作,在访问第一个网页时,需要执行建立数据链接的代码段,只要不退出浏览器,后续的访问网页就不再执行建立数据链接的代码段。可见,访问网页这种频繁操作中包含的建立数据链接的代码段,相对于每次访问网页都要执行的网页解析、网页渲染等代码,其执行频率就不高,但是建立数据链接的代码段是一个费时的瓶颈代码段。再比如程序启动操作,也是比较频繁的操作,但它所涉及的瓶颈代码段主要是由执行频率较低的初始化代码段构成。另一方面,应用程序并不仅仅处理用户输入操作,还会实现许多其他与用户交互无直接联系的功能,这些代码中也会有执行频率比较高的代码段,但加速它们并不会对提高用户交互响应速度有直接的贡献。至于偶发的用户输入操作所涉及的瓶颈代码段,其执行频率一般比较低更是显而易见的。因此,提高用户交互响应速度与提高执行频率高的代码段的运行速度不能完全等同。

综上可知:1)启动程序操作的响应速度非常重要,是衡量一个系统的用户交互响应速度的基础性指标;2)直接把执行频率高的代码段作为瓶颈代码并不完全准确,必须对用户输入处理流程进行具体分析才能找到瓶颈代码段并加速,才能有效提高设备的用户交互响应速度。现有的嵌入式 JIT 编译器都是基于动态编译和基于执行频率选择代码段来设计的,这意味着:1)无法提高应用启动操作的响应速度(因为启动操作涉及的代码往往是第一次执行,此时还没有编译好的本地码,无法对启动程序操作中的瓶颈代码段进行加速。更有甚者,由于在这个过程中可能会启动编译流程,反而会降低启动操作的速度)。2)无法完全准确获取影响用户交互响应速度的瓶颈代码,导致某些影响用户交互响应速度的瓶颈代码段不能加速,从而影响相关的用户交互响应速度。鉴于此,本文对现有 JIT 设计思想进行如下改进:1)让 JIT 编译器可以保存上次运行时的编译结果,这样程序在每次启动时就能有编译好的本地码可用;2)向应用程序提供控制代码段选择的 API,在保证原来的按执行频率选择代码段的基础上,让应用程序也可以根据自己的逻辑和实际测试结果来选择要编译的代码段,从而可以更加准确而全面地选择到影响用户交互响应的瓶颈代码。这样,JIT 能够完全加速与用户交互相关的瓶颈代码段,有效提高用户交互响应速度。

2 加速用户交互响应的 JIT 编译器设计

本文设计的可提高用户交互响应速度的 JIT 编译器结构如图 1 所示(实线表示控制流,虚线表示数据访问流)。

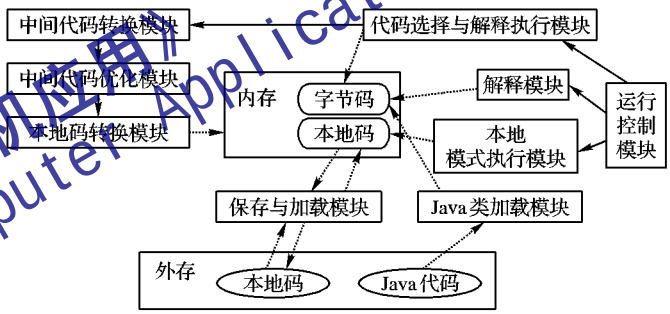

图 1 编译器的结构

其中各部分描述如下:

Java 类加载模块 把所需的 Java 类的字节码动态加载到内存中,并解析/验证出这个类的相关内容,结果存放在内存中,为使用和运行这个类做好准备。

运行控制模块 本模块是系统的控制核心。从内存中取出当前要执行的字节码,根据对应的字节码是否存在本地码,决定以本地码方式还是解释方式执行。当以本地码方式执行时,调用本地模式执行模块从内存中取出本地码执行。当以解释方式执行时,识别当前要执行的字节码是否属于可独立编译的代码段的第一条字节码,如果是,则把这条字节码交给代码选择与解释执行模块处理;否则把这条字节码交给解释模块处理。

解释模块 解释执行交给它的字节码,执行完后,返回运行控制模块。

代码选择与解释执行模块 解释执行完交给它的字节码后,判断以该字节码开始的代码段是否满足编译条件,如果不满足就返回运行控制模块;如果满足编译条件,就从内存中取出后面的字节码继续解释执行,直到遇到可能产生跳转的字节码为止。从运行控制模块传过来的字节码开始,到这个可能产生跳转的字节码结束的这段代码,就是要编译的代码段(即一段可独立编译的代码段),把这段代码交给中间代码转换模块(中间代码转换模块在另一个线程中启动编译过程),然后返回运行控制模块。由于本模块已经解释执行了一段字

节码,而不仅仅是控制模块送来的那条字节码,在返回运行控制模块时,本模块要告诉运行控制模块下一条要执行的字节码在什么位置。

本地模式执行模块　负责执行字节码对应的本地码,然后返回运行控制模块。如上所述,字节码到本地码的编译都是以代码段为单位,而不是以一条字节码指令为单位,所以执行的本地码对应的是一段字节码,因此在返回运行控制模块时,本模块要告诉运行控制模块下一条要执行的字节码在什么位置。

中间代码转换模块　这是编译流程的第一个阶段,把字节码转换成一种中间码,以利于后续处理。

中间代码优化模块　执行与 CPU 指令集无关的代码优化。

本地码转换模块　从中间码生成本地码,在生成本地码时做与 CPU 指令集相关的优化。

保存与加载模块　把在内存中的本地码保存在外存中,在应用启动时把外存中的本地码读入内存中,并据此初始化 JIT 编译器的内部状态。

根据图 1 及其各部分的描述,本文的 JIT 编译器与现有的主流 JIT 编译器结构相似,采用了同样的设计理念:基于代码段(trace)的编译,编译过程和程序执行过程分成不同线程并发运行以减少编译过程对程序执行的影响。主要不同在于:1)增加了保存与加载模块,以便能够使用上次的编译结果;2)修改了 Java 类加载模块,以解决由于使用上次的编译结果而引入的重定位问题;3)修改了本地码转换模块,以解决由于使用上次的编译结果而引入的动态解析问题;4)修改了代码选择与解释执行模块中的代码选择策略,以支持应用可以控制代码选择。

3　加速用户交互响应的 JIT 编译器实现

本文在 Andorid 2.2 版本的 Dalvik 虚拟机基础上实现了第 2 章所设计的 JIT 编译器(严格意义上说,Dalvik 执行的是一种称为 Dalvik 的字节码,它由 Java 字节码转换而来,这两种字节码指令没有本质差异,所以把 Dalvik 虚拟机看成 Java 虚拟机并不会对本文讨论有实质上的影响)。关于 Dalvik 虚拟机的具体细节,请参考文献[14],本章只讨论在 Dalvik 基础上实现第 2 章所设计的 JIT 编译器需解决的问题。

3.1　本地码转换模块

Dalvik 虚拟机在把某些字节码转换成本地码时,利用动态编译假设对生成的本地码进行了优化。在本文的 JIT 编译器中,动态编译的假设不再成立,因此 Dalvik 的本地码转换模块需要做相应修改,本文以字节码指令 OP_SGET_SHORT 为例子来说明这个修改。

OP_SGET_SHORT 的功能是获取类的某个类型为 short 的静态成员变量的值,其指令编码格式如图 2 所示。第 1 个字节是指令操作码,OP_SGET_SHORT 指令的操作码为 0x66;第 2 个字节为目的操作数,指明取得的值存放到哪个寄存器(Dalvik 虚拟机的寄存器);第 3 和第 4 个字节是源操作数,指明是哪个静态成员变量(用其索引值表示)。Java 在执行访问某个变量的操作时,需要先解析这个变量,因此,OP_SGET_SHORT 指令的标准执行流程转换成本地码如下所示(使用伪 C 代码表示):

```
StaticField * sfield;          /*指向要获取的变量的地址*/
vdst = 目的寄存器号;           /*从指令的第2字节取得*/
ref = 变量的索引值;            /*从指令的第3~4字节取得*/
/* 判断变量是否解析*/
sfield = (StaticField *) dvmDexGetResolvedField(methodClassDex,
ref);
if (sfield == NULL) {                    /*变量没有解析*/
  …
  /*解析变量(根据变量索引值获取变量地址),同时把解析结果缓存起来,下次再调用 dvmDexGetResolvedField(…)判断变量是否解析时就直接返回解析结果*/
  sfield = dvmResolveStaticField(curMethod -> clazz, ref);
  …
}
/*把变量的值(sfield -> value.s)保存在指定的 Dalvik 寄存器中(fp[vdst])*/
fp[vdst] = sfield -> value.s;
```

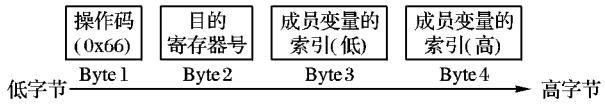

图 2　OP_SGET_SHORT 字节码指令编码格式

Dalvik 虚拟机的 JIT 编译器在生成 OP_SGET_SHORT 指令的本地码时,基于如下假设:由于这条指令是动态编译的,可以肯定它已经至少被解释执行过一次,那么变量肯定被解析了。因此,直接找到变量的存放地址,把其中的值存在目的寄存器中就可以了,其生成的本地码如下所示:

```
StaticField * sfield; /*指向要获取的变量的地址,此时 sfield 的值已经在解释执行时被赋值*/
vdst = 目的寄存器号;          /*从指令的第2字节取得*/
/*把变量的值(sfield -> value.s)保存在指定的 Dalvik 寄存器中(fp[vdst]),因为 sfield -> value.s 是常量,这个赋值语句在最终的机器码是用一个常量赋值指令来实现,进一步提高了速度*/
fp[vdst] = sfield -> value.s;
```

当允许本次运行使用上次编译的本地码时,以上假设不成立,应该生成 OP_SGET_SHORT 指令的标准执行流程所对应的本地码,这比起基于动态编译假设条件生成的本地码来说,这种方式生成的本地码要复杂得多,运行速度也更慢,因此直接按标准执行流程生成本地码会降低编译质量。本文采用一种称为 trampoline 技术[15]来解决这个问题,生成的本地码如下所示:

```
StaticField * sfield; /*指向要获取的变量的地址,该变量放在一个固定地址处*/
Label 0: Jump 跳转表的值;
Label1:
  ref = 变量的索引值;          /*从指令的第3~4字节取得*/
  /* 判断变量是否解析*/
                sfield = (StaticField *) dvmDexGetRe-solvedField
                    methodClassDex, ref);
  if (sfield == NULL) {                  /*变量没有解析*/
  /*解析变量(根据变量索引值获取变量地址),同时把解析结果缓存起来,下次再调用 dvmDexGetResolvedField(…)判断变量是否解析时就直接返回解析结果*/
  sfield = dvmResolveStatic-Field(curMethod -> clazz, ref);
  …
  }
  vdst = 目的寄存器号;          /*从指令的第2字节取得*/
  fp[vdst] = sfield -> value.s;
  跳转表的值 = Label2;
Label2:
  vdst = 目的寄存器号;          /*从指令的第2字节取得*/
  /*此时,sfield 已经是有效值,把变量的值(sfield -> value.s)保存在指定的 Dalvik 寄存器中(fp[vdst]),因为 sfield -> value.s 是常量,这个赋值语句在最终的机器码是用一个常量赋值指令来实现,进一步提高了速度*/
```

```
fp[vdst] = sfield -> value.s;
```

其中:Label1 所示的代码段是变量未解析情况下 OP_SGET_SHORT 指令的本地码,Label2 的代码段是变量已经解析情况下 OP_SGET_SHORT 指令的本地码。这两个本地码之间通过 Label0 所示的跳转指令来切换,跳转指令根据跳转表的值来控制。在初始化阶段,这个跳转表的值被初始化成 Label1 的值,当确认变量解析后,把跳转表的值改成 Label2 的值。通过 Trampoline 技术,既保证了动态解析的支持,又保证了一旦动态解析完成,又跟动态编译条件满足情况下生成的本地码一样的执行效率(只多了 Label0 所示的那条跳转指令)。

3.2 Java 类加载模块

Dalvik 的 JIT 编译器通过查询哈希表来判断字节码是否被编译成本地码:以字节码在内存中的存放地址为 Key 值,查找一个哈希表,如果表中存在与该 Key 值对应的表项,这个表项中的 Value 值就是字节码对应的本地码的存放位置(相对本地码内存区起始位置的偏移);如果表中没有与该 Key 值对应的表项,说明字节码没有被编译成本地码。因此,必须保证每次运行时,字节码在内存中的存放地址保持一致。对于本地码,因为使用的是偏移,因此本地码的内存存放地址不要求保持一致。

字节码在内存中的存放地址由 Java 类加载模块决定。Dalvik 把许多类的字节码都集中放在一个类文件的连续空间内,系统中的类文件很少;此外,类加载模块使用 Linux 操作系统的 mmap API 把类文件中的所有字节码一次映射到内存中。根据 Dalvik 的这些特点,只要保证类文件每次映射到内存时,映射的内存起始地址一样就可以让字节码在内存中的存放地址保持一致。鉴于此,我们把每个类文件映射的内存起始地址保存到一个文件中,在每次开机和应用启动时,先打开这个文件,把对应的类文件映射到指定的内存地址上,从而保证每次应用运行时,字节码在内存中的存放位置都保持不变。

3.3 保存与加载模块

保存与加载模块在保存编译结果时,不但要保存本地码,还要保存一些管理信息,例如 3.2 节中的哈希表、类文件的内存映射地址等,最关键的是对要保存的本地码中的 Predicted 类型的代码段进行解链操作。Predicted 类型的代码段对应了 Java 的方法调用的多态性:完全一样的方法调用指令,根据发出这个方法调用指令的调用者不同,会执行不同的方法。Dalvik 采用了如下所示的 PredictedChainingCell 数据结构来处理方法调用的多态性:

```
typedef struct PredictedChainingCell {
  u4 branch;
  ClassObject *clazz;
  Method *method;
  u4 counter;
} PredictedChainingCell;
```

其中:branch 指向要调用的方法对应的本地码的存放地址,clazz 表示发出这个方法调用的对象,method 表示要调用的方法信息(如名称、参数等),counter 主要涉及性能优化(在此省略)。当执行到多态方法调用指令时,查询对应的这个数据结构,首先核对要调用的方法是否跟 method 一样,然后再看 branch 是否有效(即该方法是否已编译成了本地码),最后需要看 clazz 是否跟当前发出这个方法调用指令的 Java 对象一致(通过运行时堆栈可以获得当前正在执行的 Java 对象)。如果不一致,即使 method 和 branch 满足条件也不能去执行 branch 所指向的本地码。因为 branch 所指向的本地码,是跟 clazz 相关的方法对应的本地码,而目前调用者不是 clazz,尽管要调用的方法是一样的,根据多态性,其实现体却可能不同,直接调用 branch 所指向的本地码,实际上就去调用对应 clazz 的方法实现体,导致运行错误。所有的 PredictedChainingCell 类型的变量的值是直接存放在本地码中的。很显然,当再次运行时,clazz 和 method 指向的地址都是无效的,因此在保存本地码时,必须把这些 PredictedChainingCell 类型的变量的值全部找出来,并把其中的 branch、clazz、method、counter 等数据域初始化成无效值,以免运行时出错。最后,保存与加载模块在加载上一次保存的编译结果时,由于已经有了编译结果,必须要按照保存的编译结果初始化 JIT 编译器,而不是初始化成默认值。例如哈希表、本地码内存区的内容等。

3.4 代码选择与解释执行模块

代码选择与解释执行模块的一个重要功能是决定哪些代码段可以被编译成本地码。Dalvik 使用一个执行频率哈希表来决定哪些字节码需要编译。首先,以代码段的第一条字节码指令的内存地址为 Key 值,查询一个哈希表,该表项中 value 值对应了该代码段的执行频率。如果执行频率达到阈值,就启动选择和编译这个代码段。在此基础上,我们需要增加一个与原条件成“或”关系的条件:如果不满足频率要求,就看系统是否打开了一个强行 JIT 编译的标记变量,如果是,则继续启动选择和编译这个代码段。

为了控制强行 JIT 编译的标记变量,向应用程序提供 API 接口来设置这个标记变量为 ture 或 false。这样,应用程序在自己的瓶颈代码段之前调用 API 把这个变量设置为 true,在自己的瓶颈代码段之后调另一个 API 把标志变量设置为 false。就可以保证这两个 API 之间的被执行的字节码能被 JIT 编译,从而达到 JIT 加速的效果。

4 实验结果与分析

在一个 600 MHz 的 CPU,512 MB 内存的手机上进行了对比实验,对比实验以应用启动速度这个基础指标为例。实验结果如表 1 所示。

表 1 应用启动时间 s

应用名称	启动时间	
	改进前	改进后
DocstoGo	2.2	1.3
照相机	3.0	2.5
收音机	1.0	0.5
浏览器	2.5	1.3
计算器	1.0	0.4
闹钟	1.0	0.4
拨号器	1.5	0.8

表 1 中的启动时间是指在主菜单界面上点击相应的应用程序图标到出现应用程序的第一个界面的时间。另外,每个应用每种情况下的启动操作重复 6 次,取其平均值。改进后是指把应用启动流程的瓶颈代码进行了 JIT 加速。从表 1 可看出:本文的 JIT 改进导致了应用启动速度的极大提高,除了照相机应用,平均能够让让应用启动速度提高一倍,时间缩短一半。对于照相机应用启动加速效果不佳的问题,我们仔细分析了照相机的启动流程代码,发现主要有两个因素:1)这个应用程序启动流程中很多操作都是用 JNI 来实现的,也就是说它本来就是用本地码实现的,JIT 没有加速空间;2)应用等待硬件初始化好后才显示第一个界面,而这个硬件初始化

(下转第 834 页)

机运行过程中行为特性的关键作用域,用户对该域的操作会直接影响虚拟机系统的特性。在本模型的设计中,考虑到用户可能由于错误配置而导致虚拟机系统的故障,对用户能够参与配置的选项进行了细致划分,并设计了相应的验证环节,如果设定的值不在允许的范围内,是不会最终写入配置文件,并被虚拟机加载使用的,这就保证了该模型的正确运行和整个系统的可靠性。

4 结语

测试结果表明,该模型可以减少用户在改变虚拟机执行环境运行时特性时的配置时间,在多次配置时的优势比较明显,该模型是有效的。整体而言,该模型满足了用户对虚拟机执行环境的多样性需求,在配置方式上灵活高效,提高了 VMM 的可用性。该模型虽然考虑到了对系统安全性的影响,在设计上已经对用户提供的配置参数做了验证,但是由于该接口将关系虚拟机运行特性的关键控制域在一定程度上暴露给用户,这就在一定程度上带来了安全上的隐患。进一步验证该模型的安全特性,将是本文进一步的工作。

参考文献:

[1] FISHER - OGDEN J. Hardware support for efficient virtualization [EB/OL]. [2010-07-19]. http://www.cse.ucsd.edu/~jfisherogden/hardwareVirt.pdf.

[2] NEIGER G, SANTONI A, LEUNG F. Intel virtualization technology: Hardware support for efficient processor virtualization[J]. Intel Technology Journal, 2006, 10(3): 167 - 177.

[3] 英特尔开源软件技术中心,复旦大学并行处理研究所. 系统虚拟化:原理与实现[M]. 北京:清华大学出版社, 2009.

[4] University of Cambridge Computer Laboratory. Xen architecture overview[EB/OL]. [2011-08-10]. http://wiki.xen.org/xenwiki/XenArchitecture?action=AttachFile&do=get&target=Xen+Architecture_Q1+2008.pdf.

[5] KIVITY A, KAMAY Y, LAOR D. KVM: The Linux virtual machine monitor[EB/OL]. [2010-05-10]. http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[6] WANG XIAOLIN. Dynamic memory paravirtualization transparent to guest OS[J]. Science China: Information Sciences, 2010, 53(1): 77 - 88.

[7] WANG XIAOLIN. Selective hardware/software memory virtualization [C]// VEE'11: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. New York: ACM Press, 2011: 217 - 226.

[8] Cambridge University. Xen 源代码[EB/OL]. [2011-05-16]. http://www.xensource.com.

[9] 石磊. Xen 虚拟化技术[M]. 武汉:华中科技大学出版社, 2009.

[10] GUO YUDONG, WANG XIAORUI. A cooperative model virtual-machine monitor based on multi-core platform[C]// Proceedings of the 2nd International Conference on Future Computer and Communication. Piscataway, NJ: IEEE Press, 2010: 802 - 807.

[11] 王晓睿. 虚拟机监控器体系结构研究[D]. 郑州:信息工程大学, 2010.

[12] Intel corporation. Intel 6 4 and IA - 3 2 architecture software developer's manual, Vol 3B[EB/OL]. [2011-07-30]. http://www.intel.com/Assets/PDF/manual/253669.pdf.

[13] 叶茂. 基于 VT-x 的处理器虚拟化技术研究[D]. 郑州:信息工程大学, 2010.

(上接第 826 页)

包含有机械动作(镜头对焦),花费的时间比较长,达到 1.2 s 左右,这段时间 JIT 也是无法加速的。

5 结语

本文分析了移动设备用户交互响应的特点和现有 JIT 编译器技术在此方面的缺陷,设计了一种新的 JIT 编译器技术,可以保存本次编译结果供程序下次运行时使用,并且让应用程序可以根据自己的逻辑和实际测试结果来选择要编译的代码段。在 Dalvik 虚拟机上实现了这个技术,最后通过对应用启动速度的实验分析,证明该技术可以让 JIT 完全加速与用户交互相关的瓶颈代码段,从而有效提高用户交互响应速度。最后,本文提出的技术不影响现有的 JIT 编译器技术,现有的 JIT 编译器通过改进都可以支持该技术。

参考文献:

[1] Google Inc. What is Android [EB/OL]. [2011-06-01]. http://developer.android.com/guide/basics/what-is-android.html.

[2] 王会进, 龙舜. Java 性能优化综述[J]. 小型微型计算机系统, 2008, 29(4): 720 - 725.

[3] MANJUNATH G, KRISHNAN V. A small hybrid JIT for embedded systems[J]. ACM SIGPLAN Notices, 2000, 35(4): 44 - 50.

[4] 杨博, 王鼎兴, 郑纬民. 一个基于混合并发模型的 Java 虚拟机[J]. 软件学报, 2002, 13(7): 1250 - 1256.

[5] da SILVA A F, COSTA V S. An experimental evaluation of Java JIT technology [J]. Journal of Universal Computer Science, 2005, 11(7): 1291 - 1309.

[6] CIERNIAK M, LI W. Just-in-time optimizations for high-performance Java programs[J]. Concurrency: Practice and Experience, 1997, 9(11): 1063 - 1073.

[7] BADEA C, NICOLAU A, VEIDENBAUM A V. A simplified Java bytecode compilation system for resource-constrained embedded processors[C]// CASES'07: Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. New York: ACM Press, 2007: 218 - 228.

[8] DEBBABI M, GHERBI A, KETARI L, *et al.* A synergy between efficient interpretation and fast selective dynamic compilation for the acceleration of embedded Java virtual machines[C]// PPPJ'04: Proceedings of the 3 rd International Symposium on Principles and Practice of Programming in Java. New York: ACM Press, 2004: 107 - 113.

[9] GAL A, PROBST C W, FRANZ M. HotpathVM: An effective JIT compiler for resource-constrained devices[C]//VEE'06: Proceedings of the 2 nd International Conference on Virtual Execution Environments. New York: ACM Press, 2006: 144 - 153.

[10] 史晓华, 金茂忠. 即时编译器中的代码消极生成机制[J]. 计算机工程, 2008, 34(1): 47 - 49.

[11] SHI X H, JIN M Z, CHENG B C, *et al.* Design a high-performance just-in-time compiler for a J2ME JVM on XScale[C]// ICESS'08: Proceedings of International Conference on Embedded Software and Systems. Washington, DC: IEEE Computer Society, 2008: 439 - 446.

[12] BRANDNER F, THORN T, SCHOEBERL M. Embedded JIT compilation with CACAO on YARI[C]// ISORC'09: Proceedings of the 2009 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. Washington, DC: IEEE Computer Society, 2009: 63 - 70.

[13] Google Inc. Designing for responsiveness [EB/OL]. [2011-06-15]. http://developer.android.com/guide/practices/design/responsiveness.html.

[14] Google Inc. Dalvik — Code and documentation from Android's VM team [EB/OL]. [2011-07-24]. http://code.google.com/p/dalvik.

[15] 闫伟, 谷建华. Java 虚拟机即时编译器的一种实现原理[J]. 微处理机, 2007, 29(5): 58 - 60.