嵌入式Linux库裁剪技术分析与改进*)

卢延云　孙玉芳
(中国科学院软件研究所开放系统与中文信息处理中心　北京100080)

摘　要　系统裁剪是嵌入式Linux系统开发过程中的一个重要环节。现在有很多裁剪技术可以使系统变小，本文则分析了其中的共享库裁剪技术的原理。针对现有技术的不足，本文提出并实现了一种函数级的库裁剪方案，可以在很大程度上提高裁剪效率。

关键词　嵌入式设备，Linux，共享库，裁剪，函数

The Analysis and Improvement of Library Reduction Technology for Embedded Linux System

LU Yan-Yun　SUN Yu-Fang
(Open System & Chinese Information Processing Center, Institute of Software, Chinese Academy of Sciences, Beijing 100080)

Abstract　It's an important task to reduce the system size in the development of embedded linux system. To achieve this goal there exist many technologies, among which this paper analyzes the library reduction technology. Against the deficiency of the technology in existence, this paper introduces and implements a function-level library reduction scheme, which can quite improve the reduction efficiency.

Keywords　Embeded device, Linux, Shared library, Reduction, Function

1. 引言

Linux已经越来越广泛地应用于各种嵌入式设备中。但是一般的Linux发行版都非常庞大，很难用于只有有限存储空间的嵌入式设备，所以我们必须对Linux系统进行裁剪。Linux系统大致有以下4种主要的裁剪技术，使用这些技术可以有效地减小系统的尺寸且不会影响系统的性能。

① 删除冗余文件。一般的Linux发行版中都包含很多帮助文档、辅助程序、配置文件和数据模板，在嵌入式系统中这些文件都是不必要的，完全可以删除。甚至连配置文件中的大量注释也都可以被去掉。

② 共享库裁剪。嵌入式系统的应用程序是有限的，共享库中就可能有很多永远不会被用到的冗余代码，这些代码就可以被删除。

③ 采用具有同样功能的替代软件包。Linux上有许多具有相似功能的软件包，可以选择其中占存储空间较小的软件包并其移植到嵌入式设备上，用来代替原来占空间较大那些的软件包。

④ 修改源码。包括重新配置、编译软件包，去掉不需要的功能；增加软件的模块性，从而有利于提高裁剪效率；重新配置内核，去掉不需要的驱动和模块。

以上裁剪技术中，只有共享库裁剪容易用软件实现，做成自动裁剪工具。下面我们将重点分析现有的共享库裁剪技术，根据它的不足，本文提出一种更加有效的裁剪方案。

2. 现有共享库裁剪技术的原理

库中保存着预先编译好的目标代码，一般是被应用程序反复使用的公用代码。在Linux系统中，应用程序与库之间可以静态链接或动态链接。静态链接时，链接器从库中选取应用程序需要的代码，然后复制到生成的可执行文件中。显然，当静态库被多个程序使用时，磁盘上、内存中都是多份冗余拷贝。动态链接时，链接器并不真的把库代码复制到可执行文件中；仅当可执行文件运行时，加载器才检查该库是否已经被其它可执行文件加载进内存，如果内存中不存在才从磁盘上加载该库。这样多个应用程序就可以共享库中的代码的同一份拷贝，节约了存储空间。这也是嵌入式Linux系统使用共享库的主要原因。

当使用静态链接库时，链接器会自动地只把库中被使用的模块链接到可执行文件中。但是这种方法没有用在共享库中，主要是因为在应用程序执行之前链接器并不知道应用程序最终用到了库中的哪些部分。因此要对共享库进行裁剪必须先分析动态链接的原理。

共享库和可执行文件中都有若干个符号表，其中定义了一些外部符号，分为导出(export)符号和导入(import)符号这两种。导出符号是指在该文件中定义但可以被其它文件使用的符号，一般是可以由其它文件调用的函数；导入符号是指被该文件使用了但并没有定义的符号，一般是被该文件调用的函数，而且导入符号一般指明了定义该符号的共享库。加载器在加载可执行文件或共享库之前会先遍历它的每个导入符号，检查该符号的相关代码是否已在内存中，否则先查找并加载定义该符号的共享库。由于嵌入式Linux系统中的应用程序和共享库一般都是确定的，共享库中就可能存在永远不会被别的文件调用到的导出符号，将这些符号的相应代码从共享库中删除不会影响到系统的正常运行。

现有裁剪技术都是以上述原理实现的。下面则具体分析它的实现方法。其中用符号Ψ代表某个需要裁剪的共享库。

*)本文得到国家自然科学基金项目60073022、国家863项目8633-306-2D12-14-2和中科院知识创新重大项目KGCX1-09的资助。卢延云　硕士研究生，研究方向：系统软件。孙玉芳　博士生导师，研究员，研究方向：系统软件，中文信息处理，大型数据库与网络工程。

2.1 查找 Ψ 中可能被应用程序和其它共享库用到的导出符号

用 nm、readelf 或 objdump 这些工具可以读出二进制文件(共享库或应用程序或其中间目标文件)中的所有外部符号的信息。对 Ψ 中的每个导出符号,通过遍历其余所有的共享库和所有的应用程序的导入符号就可以知道该符号是否被调用到,如果被用到就将其加入符号集合 Σ 中。

2.2 查找提供这些符号的目标文件并重新链接成较小的共享库

由于 Ψ 是由若干个目标文件链接而成的,它的导出符号都是在其中某个目标文件中定义的。由此可见,假如 Ψ 在某个目标文件中定义的所有导出符号都不在 Σ 中,我们就有可能排除该目标文件,不将其链接到 Ψ 中。当然由于各个目标文件之间还有一些内部依赖关系,将要链接到 Ψ 中的目标文件所依赖的目标文件也就不能被排除。

每个目标文件都有自己的导入符号和导出符号,若一个目标文件的导入符号是另一个目标文件的导出符号,就称前者依赖于后者。可以建立一张图,Ψ 的每个目标文件对应图上的一个结点,目标文件间的依赖关系对应于图上的一条有向边。对于每个目标文件,从图上它的结点出发的所有有向路径上的所有结点对应的目标文件就是它所依赖的。

找到所有被依赖的目标文件后,可以用事先保存的链接参数或事先指定的参数将它们重新链接起来,生成尺寸较小的共享库。

2.3 重复前两步操作直到裁剪不能继续进行为止

对所有共享库进行上述裁剪后,由于 Ψ 的导入符号也都是在它的某个目标文件中定义的,某些导入符号可能就会因为某个目标文件被排除而不再存在。那些符号就有可能不再被调用到,定义了它们的共享库就可以被进一步裁剪。重复前两步操作直到所有共享库的尺寸不再减小,裁剪才算全部完成。

3. 函数级裁剪技术

上述库裁剪技术的粒度只达到了目标文件级,未被裁剪掉的目标文件中还是有不必要的导出符号及代码。下面我们提出一种函数级裁剪技术,可以将所有不必要的导出符号及冗余代码裁剪掉。

由于导出符号也可以被 Ψ 的内部代码所调用,所以我们可以删除的导出符号不仅不能被应用程序和其它共享库所调用,也不能被 Ψ 的内部代码所调用。我们将 Ψ 的代码按源码的组成方式划分成函数块,并建立导出符号与函数块的对应关系。然后如图1所示创建一个依赖关系图 $\mathscr{R}$,为每一个函数块 F_i 在图上建立一个结点 N_i。假如函数块 F_i 调用了函数块 F_j,我们就称 F_i 依赖于 F_j,并且在图上建立一条由 N_i 指向 N_j 的有向边 $N_i \rightarrow N_j$。如果函数块 F_i 被别的文件所调用,则被 F_i 所直接或间接调用的函数块也都不能从 Ψ 中删除。在 $\mathscr{R}$ 上就是指 N_i 从开始的所有有向路径上的所有结点都应该被保留。遍历 Σ 中的每个符号,将该符号所对应的结点作上标记,并将标记的结点集合记为 Φ。然后用如下规则标记所有被依赖的结点:$\forall N_i \in \Phi, N_j \notin \Phi$,若 $N_i \rightarrow N_j \in \mathscr{R}$,则 $\Phi = \Phi \cup \{N_j\}$。这样,$\mathscr{R}$ 上剩余的未被标记的结点就是冗余的,它们所对应的函数块完全就可以从 Ψ 中删除。如图2,$\Sigma = \{N_2\}$,经过递归标记,最后得到 $\Phi = \{N_2, N_4, N_5, N_7\}$,所以函数块 F_1、F_3、F_6 就可以从 Ψ 中删除。

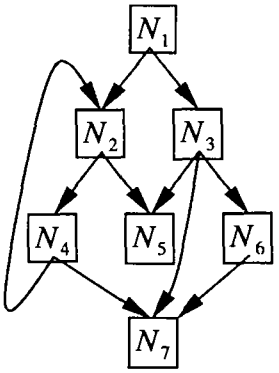

图1 一个依赖关系图

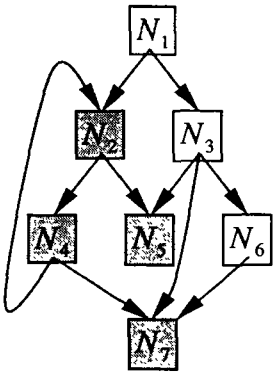

图2 标记后的依赖关系图

3.1 划分函数块

Linux 中库的源码一般是用 C 等高级语言编写的,语法比较复杂,对其进行分析也比较困难。而汇编语言的语法则简单得多,我们可以将 Ψ 的代码段反汇编并对其进行分析。以x86体系结构为例,我们可以很容易地找出如下的每个函数块:

```
00000000: <symbol>:
       0: 55       pushl %ebp
       1: 89 e5    movl %esp,%ebp
       .......
       d: c9       leave
       e: c3       ret
```

其中 symbol 是该函数块的函数名,可能是内部符号或外部符号。每个函数块就是这样以符号<symbol>开始,以指令 ret 结束。一般情况下,反汇编结果中的函数块与源码中的函数是一一对应的。当然也有不一致的时候,比如高级语言的嵌套函数,但是这并不影响我们下面对相应函数块的处理结果。

3.2 建立依赖关系

接下来要对每个函数块进行语法解析,查找该块对其它函数块的调用关系。对 C 语言的语法进行分析我们可以知道,在所有的控制转移语句中只有间接 goto 语句、函数调用和函数返回可以将控制流从一个函数块转到另一个函数块中,它们分别对应于汇编语言中的 jmp、call 和 ret 指令。而且其中只有 jmp 和 call 是我们需要的。所以我们只需要分析每个函数块中的这两个指令,找出所调用的函数块,并在 $\mathscr{R}$ 上建立相应的有向边。

但 jmp 和 call 是可以间接寻址的,这种情况下一般只有在代码执行过程中才能知道确切的跳转地址。而我们处理的代码都是只能在嵌入式设备上运行的,它的体系结构与正在执行裁剪程序的主机一般都有很大差别,因此我们不可能通

过运行代码的方式来找到跳转地址，只能采取不同的方法来处理。

C语言中只有函数指针调用语句在编译后才能生成间接寻址的call语句；间接goto语句则会生成间接寻址的jmp语句。但后者只有在将局部标号值用参数传递或用全局变量保存时才能在函数块之间跳转。而这样做会产生不可预测的结果，因此一般的C代码中不会有这样的语句，我们也就不考虑这种情况。我们认为间接寻址的jmp语句只能将控制流转移到本函数块的其它地方，并不会产生函数块之间的调用关系。

现在我们只需要考虑C语言中的函数指针。函数指针是指向某个函数块起始地址的变量，它的值不可能无中生有，必须有某个语句引用该函数块的函数名得到所需地址后再通过某种方式将值传给函数指针；否则该地址只能是程序员事先知道的Linux核心中的某个例程的起始地址，我们的依赖图中并没有该例程的对应的结点。假设函数块 F_a 通过函数指针调用了函数块 F_b，我们认为必须有一条语句取得 F_b 的起始地址并最终将该地址传给了 F_a 中的函数指针，并假设该语句位于函数块 F_c 中。那么在 $\mathscr{R}$ 中我们用一条“弱”的有向边 $\xrightarrow{N_c}$ 连接结点 N_a 和 N_b，表示只有 N_c 被标记后 N_a 和 N_b 之间的依赖关系才存在。这是因为，如果 N_c 没有被标记，F_c 根本不会被执行，F_a 中的那个函数指针就不会指向 F_b 的起始地址，F_b 也就不可能被 F_a 所调用。

代码的实际执行情况会复杂得多，包括 F_c 在内，引用 F_b 的函数名并取得它的起始地址的函数块可能有很多个，也不是每个这样的函数块都会将起始地址传递给 F_a 的函数指针。因此很难确定 N_a 和 N_b 之间的“弱”有向边的条数及每条边所依赖的函数块。但是由于每个函数块中是否引用了其它函数块的函数名是很容易确定的，假如我们用 N_c 到 N_b 的有向边代替 $N_a \xrightarrow{N_c} N_b$ 得到依赖图 $\mathscr{R}'$，并且能够证明 $\mathscr{R}'$ 比 $\mathscr{R}$ “强”，也就是说对于同样的 Σ 集合，$\mathscr{R}'$ 的标记集合 Ψ 总是包含 $\mathscr{R}$ 的标记集合 Φ，那么我们就可以消除 $\mathscr{R}$ 中所有的“弱”有向边而不会使 $\mathscr{R}$ 中应该被标记的结点变成未标记的。这样可能会额外标记少量结点，但是从很大程度上减少了建立依赖图的难度。

由于我们变换的有向边只涉及到 N_a、N_b 和 N_c 这三个结点，且有向边的终点都是结点 N_b，所以变换只会影响到 N_b 及其所依赖的结点是否被标记。我们只需证明不论 N_a 和 N_c 是否被标记，若 $N_b \in \Phi$，则 $N_b \in \Psi$。如果 $N_b \in \Sigma$，显然不管在 $\mathscr{R}$ 还是 $\mathscr{R}'$ 中，都有 $N_b \in \Phi$。以下只需在 $N_b \notin \Sigma$ 的前提下考虑图3中的四种情况：

①$N_a, N_c \notin \Phi$。

$\because N_b \in \Phi, \therefore \exists\ N_d \in \Phi, N_d \rightarrow N_b \in \mathscr{R}$。则 $N_d \in \Psi$ 且 $N_d \rightarrow N_b \in \mathscr{R}'$。

$\therefore N_b \in \Psi$

②$N_a \in \Phi, N_c \notin \Phi$。

同①

③ $N_a \notin \Phi, N_c \in \Phi$

$\because N_c \in \Phi, \therefore N_c \in \Psi$。又$\because N_c \rightarrow N_b \in \mathscr{R}', \therefore N_b \in \Psi$

④ $N_a \in \Phi, N_c \in \Phi$

同③

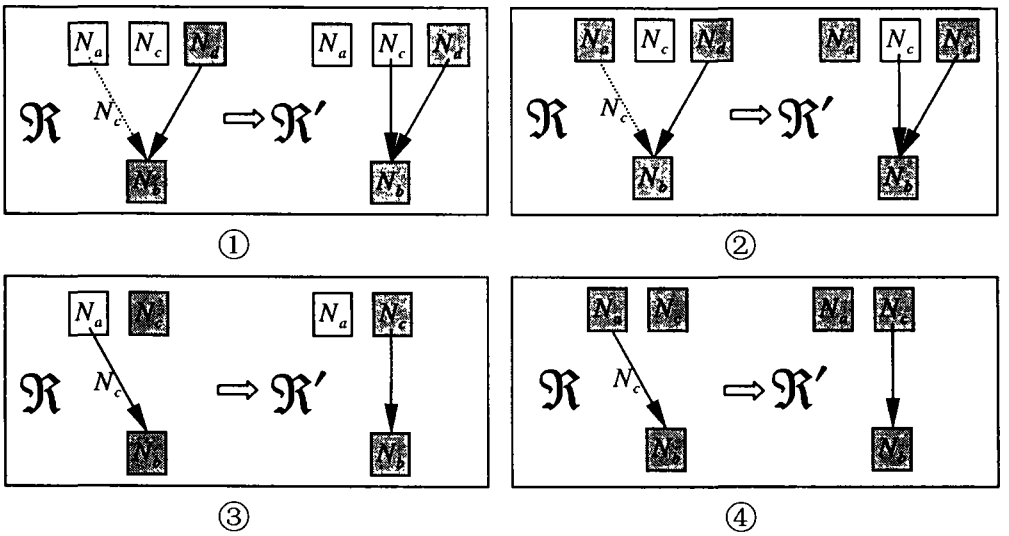

图3 $\mathscr{R}'$ 比 $\mathscr{R}$ 强

因此，现在我们只有在以下两种情况下才需要在 $\mathscr{R}$ 上建立有向边 $N_i \rightarrow N_j$：

① 函数块 F_i 引用函数块 F_j 的函数名；

② 函数块 F_i 中直接寻址的call指令将控制流转到函数块 F_j 中。

3.3 对需要保留的函数块进行标记，裁掉未标记的函数块

用下面的深度优先的递归算法可以将 $\mathscr{R}$ 上以 Σ 中每个元素开始的所有有向路径上的所有结点都作上标记：

```
Procedure 对结点 N_i 进行递归标记
{
    if N_i 已被标记 then 退出递归过程
    标记 N_i
    对从 N_i 开始的每个有向边 N_i→N_j，对结点 N_j 进行递归标记
}
对 Σ 中每个结点 N_i 进行递归标记
```

标记了所有要保留的函数块后，有两种方法可以从 Ψ 中去掉某个未标记的函数块。一种方法是从源代码中将函数删除，然后重新编译和链接成共享库；另一种方法则是从二进制文件中将冗余代码删除。前者实现简单，但基本上只能手工实现；后者则可以用程序实现自动裁剪。但从代码段去掉一段代码后，后面的代码需要往前移，所有相关指令中的地址都得重

（下转第169页）

SoftOS 的 PDM 模块优点在于：1)多个管理员共同管理系统，相互牵制，更加符合最小特权的安全原则；2)双重安全检查，系统管理工具(属于固定的系统资源，其完整性能够得到保证)是由操作系统厂商提供，一般实现了相应的安全检查机制，此外核心在安全管理工具运行时还要进行必要的权限检查。3)表决系统有效解决个别管理因特殊原因无法实施系统管理的困难。4)加入 PDM 后对系统性能影响不大，产品化容易。

结束语 安全操作系统 SoftOS 的基于特权分化模型 PDM 能够较好地解决操作系统自保护和管理控制机制薄弱方面的缺陷，对于防止由于权力过于集中而引起的有意或无意的破坏性操作有着较好的防范作用，提高了系统的容错性。大量的测试及比较表明，加入 PDM 模块后的 SoftOS 在安全管理增强功能方面，已经达到较高水平。

参 考 文 献

1 Olson I M, et. al. Informatin Security Policy, Information Security: An Integrated Collection of Essays Edited by Marshall D. Abrams, et al. IEEE Computer Society Press, 1995

2 IEEE, IEEE POSIX. 1e Draft Standard for Information Technology: Portable Operating System Interface (POSIX): Protection, Audit, and Control Interface, IEEE Computer Society Press

3 Department of Defence, America. TRUSTED COMPUTER SYSTEM EVLUATION CRITERIA CSC-STD-001-83, Aug. 1993

4 Acharya A, Raje M, MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In: 9th USENIX Security Symposium, Aug. 2000

5 LaPadula L J, Bell D E. Secure Computer Systems: Mathematical Foundations: [MITRE Technical Report 2547]. Volume Ⅰ, 1973

6 O'Shea G. Security in Computer Operating System. NCC Blackwell Limited. 1991

7 Ware W H. Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security, R-609-1. Santa Monica CA. Rand Corp. 1970

8 Wright C, Cowan C, Linux Security Modules: General Security Support for the Linux Kernel. In: 11th USENIX Security Symposium, San Fransisco 2002

9 http://www.nsa.org/selinux/

(上接第164页)

定位。由于共享库的代码是位置无关(PIC)的，内部跳转只使用相对地址，全局函数和数据的绝对地址则是由装载器计算并填入 GOT 表中，然后代码通过该表来间接地引用所需地址。因而我们只需要重定位代码中的相对地址。

3.4 重复前三步操作直到裁剪不能继续进行为止

同2.3节一样，重复前三步操作直到所有共享库的尺寸不再减小，裁剪才算全部完成。

3.5 两种技术裁剪效果的比较

由于 libc 通常是文件系统中最大的共享库，不经任何裁剪一般大小都有1.2M 字节以上，比 Linux 内核还要大得多。所以要是能对 libc 进行一定的裁剪，整个系统的尺寸的减小就比较显著。我们就以对 libc 的裁剪效果比较两种技术的优劣。

从图4中我们可以看到，我们提出的函数级裁剪技术的裁剪效果比原来的目标文件级裁剪技术明显要好，在应用程序数量不多的时候，用前者裁剪的 libc 大小比后者要小200多 kb；在应用较多的时候也要小100多 kb。而在总体上，用我们的技术可以将 libc 裁剪掉500～800kb。由此可见，该技术的裁剪效果还是非常明显的。

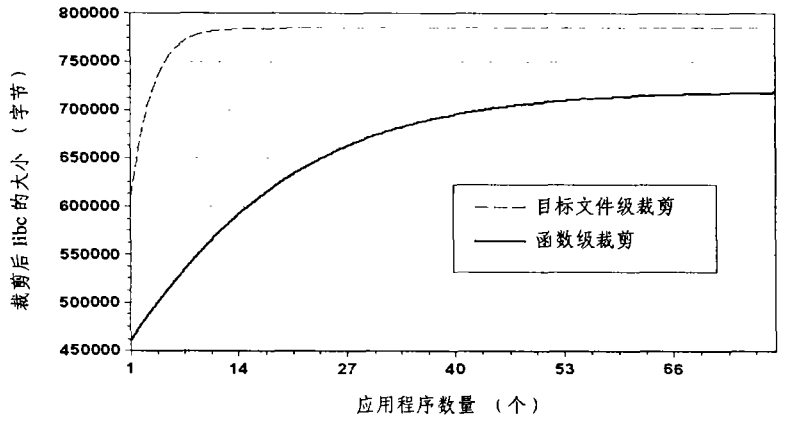

图4 两种技术的裁剪效果比较

小结 本文提出并实现的函数级的共享库裁剪技术能将库中大部分冗余代码裁剪掉，裁剪效率比原来的目标文件级裁剪技术有了很大提高。但是它还是有许多不足，包括：要求库的源码编写比较规范；不同体系结构需要有不同的处理等。但毕竟库裁剪领域才发展不久，还不是很成熟。经过对该技术长时间的测试，相信我们能够弥补它的不足，使它能够在嵌入式 Linux 领域广泛使用。

附录：本文用到的记号

Ψ：将要被裁剪的共享库；Σ：Ψ 中被用到的导出符号的集合；$\mathscr{R}$，$\mathscr{R}'$：Ψ 中各函数块的依赖关系图；F_i：Ψ 的某个函数块；N_i：F_i 在 $\mathscr{R}$ 上的对应结点；Φ，Φ'：$\mathscr{R}$ 和 $\mathscr{R}'$ 上被标记的结点集合。

参 考 文 献

1 Bird T, Right-Sizing Linux: Selective Reduction of Linux for Embedded Devices. Embedded Systems Conference, 2000

2 Levine J R. Linkers and Loaders. Morgan-Kauffman, 1999

3 Executable and Linking Format Spec v1.2. TIS Committee, 1995

4 Aho A V, Sethi R, Jeffrey D. Ullman, Compilers - Principles, Techniques, and Tools. Addison-Wesley, 1986

5 Stevens W R. Advanced Programming in the Unix Environment. Addison-Wesley, 1992

6 路丝林，朱珂等译. David Rusling, Linux 编程白皮书. 机械工业出版社，2000