Linux Kernel Customization for Embedded Systems By Using Call Graph Approach

Che-Tai Lee

Dept. of Info. Eng.
Feng Chia University
Taichung 407, Taiwan, R.O.C.
Tel:886-4-2451-7250
Fax:886-4-2451-6101
e-mail:ctlee@ultra2.iecs.fcu.edu.tw

Zeng-Wei Hong

Dept. of Info. Eng.
Feng Chia University
Taichung 407, Taiwan, R.O.C.
Tel:886-4-2451-7250
Fax:886-4-2451-6101
e-mail:stewart@ultra2.iecs.fcu.edu.t

Jim-Min Lin

Dept. of Info. Eng.
Feng Chia University
Taichung 407, Taiwan, R.O.C.
Tel:886-4-2451-7250
Fax:886-4-2451-6101
e-mail:jimmy@fcu.edu.tw

Abstract - It has been attracting attention to reconfigure a GPOS for application-specific domain. Linux is currently one of the popular candidate GPOSs. Although Linux has tools for kernel's reconfiguration by letting users add/remove desired function modules, we still lack of good schemes of reconfiguring Linux according to a specific embedded system. In this article, we are going to propose an approach to customize an application-specific Linux. This approach derives from a "call graph" based on reengineering. By analyzing a graph-structure representation of the target system, we could find the rules of removing the unnecessary codes of Linux.

I. Introduction

Recently, there existed many distributions of embedded Linux, such as IBM Linux watch [1]. Linux attracts industries' attention due to the following attributes

- The source code is free.
- Linux is able to provide enough functionality for extracting and reusing.
- Linux is robust, reliable, modularized, and configurable in nature.

However, Linux has some difficulties to be an embedded system.

- Linux is a monolithic GPOS and has diverse versions. Realizing this kind of huge OS is complicated. Therefore, we still lack of a formal and useful scheme to cope with it. After all, manually customizing a Linux is hard and costly.
- It is hard to guarantee the customized kernel's completeness.
- Although Linux also provides the ways for users to dynamically add/remove modules, the code of this method still occupies space of storage.

In this article, we propose a call graph [2,3,4,5] approach to customize Linux as an application-specific OS. Call graph is commonly employed to represent the interrelationships among the procedures in a program. According to a call graph, it might be able to extract the reusable components from a software system [6]. Hence, call graph is usually used for the purposes of software reengineering or software maintenance. Our approach is to use the call graph scheme to represent three parts of a Linux system. They are *software applications*, *system libraries*, and *Linux kernel*. Of course, it would be a good approach that a Linux engineer firstly adopts a traditional Linux tool - "make config" to remove the most likely unused functions. Then, by tracing the calling relations based on the various requirements, including hardware and software configuration, of an embedded system, it could be able to discover the unused code more precisely. And then we could remove them from Linux. Finally, a smaller and an application-specific Linux would be obtained.

In this study, we adopt a popular audio tool, ACDC [7], that is a CD player application in Linux, as the target system for demonstration. Because this CD player box is a headless system, that is a system without a display and keyboard, its user interface should be removed at the first step. Then according to ACDC software and target device, we remove the unused code, such as the unnecessary hardware drivers. Based on this experiment, we will list the related statistics to verify the feasibility and correctness of our approach. Finally, a complete and smaller customized Linux would thus be obtained.

II. A graph-based approach for customizing Linux kernel

Linux kernel and other Unix-like systems are monolithic operating systems. Linux consists of a number of procedures and these procedures cooperatively perform jobs by calling each other. However, Linux kernel is a non-fixed structure. This kind of property is different from the typical program that has a fixed hierarchy. Hence, it is more complicated to predict Linux kernel. The first challenge of customizing Linux kernel is to precisely understand its structure.

Call graph is a solution to cope with this challenge, because it has well ability to depict a program's calling structure. The notion of a call graph is to extract the calling relations of the invoked procedures. Recently, call graph has been applied on the fields of software reengineering and software maintenance. Our study adopts call graph technique to abstract a Linux kernel. The basic concept is to construct a kernel' calling structure and to remove the unnecessary codes according to specific application. Finally, our result tries to reconstruct an application-specific Linux kernel and to be ported onto the target device.

However, there might be several issues coming with

customizing Linux kernel:

- Although kernel is the main role of Linux, it is not the only one participant. Almost modern OSs are constructed as layer structure and Linux is also one of them. Kernel is the medium layer of Linux. Other layers such as application, library and device drivers should also be reusable.
- You can't find a rooted procedure (a rooted procedure is similar to the main(), which is a root of a C program) of Linux kernel. Hence, the calling relations among kernel's procedures are intricate.

Our stepwise approach has 7 steps including constructing Linux application's call graph and library's call graph. This approach's advantage is to reuse each layer's asset. The rest parts of this section will indicate each step's details.

Step 1. Construct an application's call graph from the application source code

Linux is an open-resource OS and supports diverse valuable applications, packages and device drivers. Moreover, you can acquire the source code easily, especially from Internet. This attracts many embedded system companies' attention, because they strongly believe that it would be an efficient way to manufacture distributions from Linux.

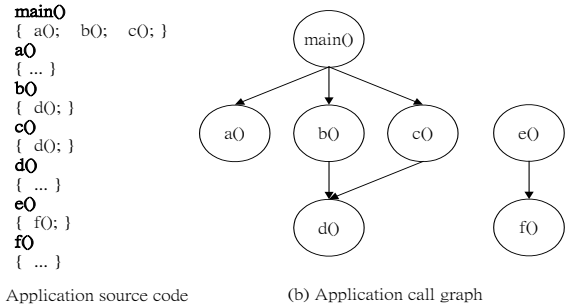

Fig.1. Generating a call graph from the application source code

The first step of the approach is to construct an application's call graph, as the source code is available. Recently, almost applications running on Linux are written in C language. Based on the concept of call graph, we can easily construct this application's call structure.

Definition. A call graph is a directed graph, A call graph of a program is formally defined as a graph *CG=(P, E, s)*, where $P \bigcup s$ is the set of nodes (the nodes represent the procedures of this program), and the set of *E* is defined by the call relations on $(s \bigcup P) \times P$, i.e., a directed edge $(p1, p2) \in E$ exist *iff p1* calls *p2* one or more times.

According to the definition, a call graph represents a program's static structure. Take a C program as the example, you can find the path composed by a number of invoked procedures and *main ()* is the starting node of this path (figure 1(b)). SUCC(main) should contain the necessary procedures. Hence, if there existed a procedure $P \notin$ SUCC(main), *P* is the unused procedure of this program. Consequently, it is better to remove *P* from this program. This is an interesting question: why did there exist an unnecessary procedure? This situation usually appears due to the programmer's mistake, especially in a huge program like Linux. Although the procedure *P* has not caused any fault within Linux, we can't make sure it is safe when porting it onto the embedded system. In this case of figure 1, two procedures, *e()* and *f()*, have to be removed.

Step 2. Construct a Library's call graph from the library's source code

Linux is also a layer-based OS. Application is on the top layer and sends request to the next layer. Typically, a Linux application, which is written in C, requests OS's services through library calls (figure 2(a)). In Linux, libraries such as I/O functions, mathematics functions, and string functions, etc, usually occupy a lot of space. Almost embedded Linux companies prefer to develop a new library from scratch. However, based on the concept of software reuse, we believe there are many reusable library calls. If we can reuse them by an efficient way, that would reduce development time.

Therefore, the second step of this approach is to construct a library call graph. This graph also represents the library's calling structure, but it doesn't have a root. In the other words, it provides a number of entries for application written in C (figure 2(b)). The purpose of analyzing a library's calling structure is to realize which library call may be reusable. Of course, we may be able to predict the unnecessary ones.

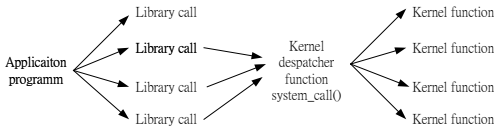

Fig. 2(a) Call a system service by library call

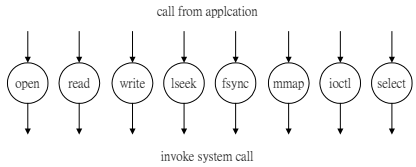

Fig. 2(b) Simplified library call graph

Step 3. Construct a kernel's call graph from the source code

Library's next layer is kernel, which is the central part of Linux. It controls and coordinates operations. Because kernel is an important and big building block, almost embedded Linux distributions prefer to reserve kernel instead of adapting it. Their basic consideration is to guarantee kernel's correct execution. Therefore, many unnecessary codes still reside within kernel.

The third step of this approach is to analyze kernel's calling structure. This step is similar to step 2. Kernel's calling structure supports several entries for last layer. Almost modern OSs are interrupt-driven systems. The events request the services of CPU with interrupt or

exception. Whenever an interrupt or exception occurs, kernel starts the relative handlers to cope with it. Therefore, in order to abstract Linux kernel, we firstly have to understand when kernel will be executed. We list the following occasions:

- An application invokes system calls.
- The occurrence of any exception.
- The occurrence of interrupt.

The first and second conditions are for application, and the third condition is for hardware device or platform. Therefore, in order to look for the unnecessary procedures of kernel, we can observe kernel's call graph according to the above conditions. For application, we could combine application's call graph, library's call graph, and kernel' call graph. This combination would generate a SUCC(main). Any procedure not in this set may be able to be removed.

Step 4. Identify which hardware device embedded system have and needs

Linux supports many hardware devices, such as disk, keyboard, mouse, and so on. However, an embedded system is also a specific-hardware platform. For example, a keyboard or a mouse may be not standard equipment for PDA. Therefore, to adapt Linux into an embedded OS, we have a sense that there existed too many unused device drivers or related code in Linux. These parts might occupy the embedded system's resources. Therefore, the fourth step is to seek these unused parts possibly.

Step 5. Combine application's call graph, library's call graph, and kernel's graph to extract which system calls that an application needs

From step 5 to step 7, we want to find a set of SUCC(main). This set describes a number of procedures that we are able to reuse them for a specific application. As we discuss above, a library's call graph lacks of a root. Therefore, we incorporate application's call graph into library's call graph. Now the main(), which is the root of the application's call graph, represents the unique entry of this combined call graph. Moreover, according to the SUCC(main), we could extract the unused library's calls.

As we discussed above, a library call is a channel to connect application and Linux kernel. By combining kernel's call graph and library's call graph, we can find out which system calls interact with library's calls. Hence, the invoked system calls are added into the SUCC(main). Consequently, we can extract the unnecessary system calls.

Step 6. Identify which exception handlers the kernel needs

In step 6, we try to remove the unemployed code that handles exceptions. As we know, some exception handlers are not needed. There have been about 18 handlers within Linux 1.2.3 including divide_error, debug, nmi, int3, overflow, bounds, invalid_op, device_not_available, double_fault, coprocessor_segment_overrun, invalid_TSS, segment_not_present, stack_segment, general_protection, page_fault, coprocessor_error, and alignment_check. These exceptions are also the entries provided by kernel.

Step 7. Remove the unused procedures and test the new kernel

After identifying the unnecessary procedures, we remove them from kernel in step 7. However, we must validate this new generated kernel's correction. If the new kernel has faults, we have to verify its call graph and to generate the correct graph.

III. A case study: ACDC player

In this section, we apply call graph approach on a case study called ACDC player. ACDC is a media application running on Linux. In order to demonstrate our approach, we adapt ACDC into an embedded application. We assume that a company decides to reuse ACDC and to port it onto another device. We call this final product as ACDC player.

In 3.1, we indicate how to construct ACDC's call graph. In 3.2, we list statistics to show our demonstration.

3.1. The customization of ACDC

At beginning, our approach can be summarized into three phases:

- Construct call graphs including application, library and kernel.
- Find the unused code, procedures, and drivers.
- Remove those unnecessary participants and test the new kernel.

Therefore, to reuse and customize ACDC, we firstly construct the related call graphs. Then we combine three call graphs and try to obtain the SUCC(main). In this case, our experimental environment is shown as follows:

- Linux distribution: Slackware 3.0
- Kernel's version: 1.2.3
- Library: libc 4.6.27
- Target application: text mode ACDC

Kernel 1.2.3 has been adopted on academic researches, due to its simplicity. This kernel doesn't support too complex functionalities. For an embedded domain, these functionalities are often not applicable. Hence, it would be easier to analyze this kernel's structure. ACDC running on this kernel is a text mode application. For a CD player, it doesn't need a GUI. Therefore, text mode ACDC could simplify our demonstration.

Firstly, we construct ACDC's call graph to find the necessary procedures. Some of these procedures in ACDC are provided by *libc*, which is a well-known library in Linux. These procedures can be regarded as the entries whenever Linux application requests its next layer's services. The next step, therefore, is to construct *libc*'s call graph. According to this call graph, we might remove the unused library's calls. The six libc's calls, *malloc, free, exit, close, open*, and *ioctl*

are channels between ACDC and libc. It also means the six libc's calls are included into SUCC(main). Therefore, we can remove other libc's calls.

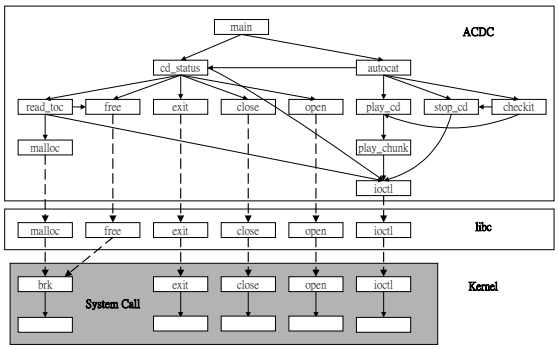

Fig. 3. The combined call graph

Figure 3 shows that kernel provides five system calls as the kernel's entries including mmap, exit, close, open, ioctl. Each of the five system calls has to call its successors. Finally, through the cooperation among these system calls, kernel serves its last layer's requests. The system calls that are not included into SUCC(main) will be removed.

3.2. Evaluation results

Linux kernel can be decomposed into several parts and each of them is stored in individual directory. Linux provides "make config' command to add/remove device drivers.

We establish two experiment cases as follows:

Case 1. The total size of kernel customized with call graph approach.

Case 2. Mix two methods, "*config*" command and call graph, to customize Linux kernel. In this case, we want to observe if a smaller kernel than first case can be generated or not.

Table 1. The statistics of case 1

Linux kernel's organization	Original size (byte)	After using call graph approach (byte)	Percentage
arch/i386/kernel/head.o	60,617	60,617	0 %
init/main.o	8,314	6,598	20.6 %
init/version.o	639	639	0 %
arch/i386/kernel/kernel.o	43,339	32,286	25.5 %
kernel/kernel.o	73,000	50,730	30.5 %
arch/i386/mm/mm.o	3,767	3,714	1.4 %
mm/mm.o	37,079	32,336	12.8 %
fs/fs.o	92,121	71,006	22.9 %
net/net.o	117,098	115,606	1.3 %
Ipc/ipc.o	22,472	21,355	5 %

Table 1shows the statistics of case 1. Through this table, our approach is able to remove 13.9% of procedures from kernel, especially on *kernel.o*, which is core of kernel. Therefore, it proves that our approach can effectively customize kernel. We almost eliminate about 30.5 % of procedures from *kernel.o*.

Table 2. The statistics of case 2

Linux kernel's organization	Phase 1	Phase 2	Percentage
arch/i386/kernel/head.o	60,617	60,617	0 %
init/main.o	8,314	6,598	20.6 %
init/version.o	639	639	0 %
arch/i386/kernel/kernel.o	36,710	32,478	11.5 %
Kernel/kernel.o	58,054	46,288	20.7 %
arch/i386/mm/mm.o	3,660	3,660	0 %
mm/mm.o	32,336	32,336	0 %
fs/fs.o	70,567	70,567	0 %
net/net.o	15,311	15,311	0 %
Ipc/ipc.o	221	221	0 %

Table 2 lists the related statistics of case 2. Case 2 contains two phases. At beginning, we adopt "config" to configure kernel as possible as we can. In the second phase, we adopt call graph approach to advance customization. Consequently, we find call graph approach removes about 20.7% of procedures from *kernel.o* after customizing with "config".

IV. Conclusions

In this paper, we try to adopt call graph representation to depict Linux kernel's calling structure. Call graph is an adaptive structure to meet different application-specific domains. This property is beneficial for embedded system design. Our approach is not like typical methodologies that customize Linux kernel with lines of code. The typical methods are inefficient and costly. An experiment called ACDC player is the demonstration. The results show that our approach can remove about 20.5% of kernel's procedures and causes no side effect.

References

[1]. http://www.ibm.com/products/gallery/linuxwatch.shtml

[2]. Hecht, M. S., *Flow analysis of Computer Programs*, North-Holland, New York, 1977.

[3]. B. Ryder, "Constructing the call graph of a program," *IEEE Trans. Software Eng., vol. SE-5*, pp. 216-225, May 1979.

[4]. J. P. Banning, "An Efficient way to find the side effects of procedure calls and the aliases of variables," *Proc. 6th Annu. Symp. Principles of Programming Languages*, ACM, 1979, pp. 29-41.

[5]. Callahan, D., Carle, A., Hall, M.W., and Kernedy, K., "Constructing the procedure call multigraph," *IEEE Trans. on Software Engineering*, SE-16(4):483-487, April 1990.

[6]. A. Cimitile and G. Visaggio, "Software Salvaging and the Call Dominance Tree," *The Journal of System and Software* 28, 1995, pp. 117-127.

[7]. http://www.hitsquad.com/smm/linux/CD_PLAYERS/