

An Application-Oriented Linux Kernel Customization for Embedded Systems

CHI-TAI LEE, JIM-MIN LIN, ZENG-WEI HONG AND WEI-TSONG LEE*

Department of Information Engineering and Computer Science

Feng Chia University

Taichung, 407 Taiwan

E-mail: jimmy@fcu.edu.tw

**Department of Electrical Engineering*

Tamkang University

Tamshui, 251 Taiwan

How to reconfigure a general purpose operating system (GPOS) into an embedded operating system has attracted attention for application-specific domains. Linux is currently one of the popular candidates for GPOSs. Although Linux has tools for kernel re-configuration by letting users add or remove desired function modules, the best schemes of reconfiguring Linux according to a specific embedded system are not practical. Even after this configuration, the target Linux might still be a GPOS. In this article, we will propose an approach to customizing an application-specific Linux operation system. This approach derives from a “call graph” based on reengineering. By analyzing a graph-structure representation of the target system, its hardware and software specifications are determined. Thus, we can find the rules for removing the redundant code in Linux. Moreover, we employ the call graph approach to verify the system integrity at the source-code level. In order to demonstrate the proposed idea, an experimental system will also be reported in this article. The results show that our approach can significantly remove about 17 percent of the Linux kernel’s footprint with respect to unreachable code.

Keywords: embedded operating system, general purpose operating system, Linux kernel customization, call graph, redundant code, unreachable code, dead code

1. INTRODUCTION

For a number of years, most prior efforts to attain embedded system involved purchasing an embedded system from a vendor and then modifying the system in collaboration with the supplier, or simply developing a new embedded system from scratch. It would appear logical to look for some way to save time by modifying a currently available system. It is in this spirit that we speak of modifying GPOS to obtain an embedded system. However, characteristics such as easy access to source code as well as having an unbounded environment for further development become the necessary criteria for selecting a GPOS of this kind. Fortunately, Linux is a GPOS which possesses such characteristics. More recently, Linux has been more frequently for use in embedded systems, for example, the TiVo digital recorder (DVR) [1], Rio MP3 player [2], and Axis Network

Received March 30, 2004; accepted June 30, 2004.

Communicated by Chu-Sing Yang.

Camera [3]. Generally speaking, Linux attracts industries' attention due to the following attributes:

- The source code is free. For developing an embedded system, this is desirable.
- Linux is able to provide enough functionality for extracting and reusing. These characteristics cover microprocessor architectures, graphics, telecommunication, and hardware devices.
- The Linux kernel aims to be compliant with the IEEE's POSIX (Portable Operating Systems Interface for Unix) standard, meaning that most existing Unix applications can be compiled and executed on a Linux system with little or no modification to its source code.
- Linux is robust, reliable, modularized, and configurable in nature.

However, reducing the code footprint as much as possible is a critical issue for embedded system developers, because the developer must take many issues into consideration, like space, weight, power consumption, and price. In that case, Linux has some drawbacks when used in an embedded system:

- Linux is a monolithic GPOS and has diverse versions. Also, adapting it is not an easy task. Therefore, there is a lack of a formal and useful scheme to cope with problems of this kind. After all, manually customizing a Linux is difficult and costly.
- It is hard to guarantee and/or test the completeness and accuracy of a customized kernel.
- Linux has provided three commands, `make config`, `make menuconfig`, and `make xconfig` for *configuration management by preprocessing* [4], for users to choose specific kernel functionalities. However, designing an embedded system using these three recommended methods may not be suitable. The system designer has to configure a new kernel according to his or her experiences, a lengthy task. Most of all, the kernel derived from this approach is still a GPOS rather than an embedded OS.
- Although Linux is configured as modules, it is a useful way for users to link and unlink object files at runtime. However, this feature may not be suitable for an embedded OS, especially for handless operation. The code size for module related software interface and data structure still occupies much storage space even though we don't need to use all the modules in an embedded system environment.

The redundant code is an existing program slice but is never reached, i.e. there is no control flow path to it from other parts of the system. Another case of redundant code is computing a value which contributes nothing, because the result is never used or is used but with no benefit to the system. The presence of redundant code in the Linux kernel may result from logical errors due to alterations in its control flow or from significant changes in the assumptions or environment of the program. Thus, the code that is redundant can be eliminated without causing any ill effects to the system. Linux is a kind of GPOS that is designed to support a variety of popular functions, nevertheless, an embedded system is designed for specific purposes and for individual use only. According to this principle, when we use a GPOS (such as Linux) as an embedded system, there will

probably be some redundant code. This redundant code will either be never executed (unreachable code) or contribute nothing (dead code) to the targeted embedded system.

Therefore, for an embedded system, it is beneficial to eliminate redundant code [5, 6]. A typical approach to eliminating redundant code from the Linux kernel and adapt it to an embedded system is through manual processing of an existing Linux kernel, say, uClinux [7]. The developer scans the kernel source code first, and then, identifies and manually eliminates the redundant code line by line. The developer can control the code size and functionality of the Linux kernel. This is probably the most useful approach to getting the smallest possible Linux kernel for a specific embedded system. However, it is not so easy for a general user, and will take lots of time to do the job. Furthermore, it should be redone again when a new version of Linux kernel is released, or when the applications or the embedded platform is changed. Developer should carefully decide which part of kernel code has to be eliminated every time the kernel customization is performed. Another drawback of this approach is that it is challenging to debug when system crashes.

In this article, we propose a call graph [8-10, 15, 16] approach to customizing Linux as an application-specific OS. A call graph is commonly employed to represent the interrelationships among the procedures in a program. Using a call graph, the reusable components from a software system can be extracted [11-13]. Hence, a call graph is usually used for purposes of software reengineering or software maintenance. Our approach is to use a call graph to represent the three parts of a Linux system, namely, *software applications*, *system libraries*, and *Linux kernel*. Of course, it is advantageous for a Linux engineer to first employ a traditional Linux tool, “make config”, to remove functions that are most likely unused. Then, by tracing the calling relations based on the various requirements, the unused code can be discovered more precisely for an embedded system, including hardware and software configuration. Afterwards, we could remove these codes from Linux. Finally, a smaller and an application-specific Linux system is obtained. In this study, we adopt a popular audio tool, ACDC [14], a CD player application on Linux, as our target system for demonstration, assuming that a user needs to manipulate a Linux CD player box. Since it is a handless system, a system without a display and keyboard, its user interface should be removed in the first step. Then according to ACDC software and target device, the unused code, such as the unnecessary hardware drivers, is removed. 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 is thus obtained.

The rest of this paper is organized as follows. Section 2 describes how to use the call graph approach to identify and eliminate unreachable code from the Linux kernel without ill effects and modify Linux kernel for application-specific system. An experimental case for the proposed approach is then reported in section 3. Finally, a brief conclusion is given in section 4.

2. A GRAPH-BASED APPROACH FOR CUSTOMIZING LINUX KERNEL

The kernel of Unix-like systems are monolithic operating systems. Linux consists of a number of procedures which cooperatively perform jobs by calling each other. In short, the Linux kernel is a non-fixed structure. The monolithic property of this kind, huge and

modifiable, is different from a typical program that has a fixed hierarchy. Hence, it becomes more complicated for a designer to predict the Linux kernel in advance. The first challenge of customizing a Linux kernel is to precisely understand its structure.

The call graph is a solution to cope with this challenge because it provides efficient capability to depict a program's calling structure. The notion of a call graph is to extract the calling relations of the invoked procedures. Recently, the call graph has been successfully applied in software reengineering and software maintenance. Our study adopts the call graph technique to abstract a Linux kernel into a kernel's calling structure and to remove the unnecessary kernel codes for a specific application. The resulting kernel will thus be an application-specific Linux kernel.

However, there might be several issues related with customizing the Linux kernel:

- Most modern operating systems are constructed using a layer structure (see Fig. 1) and Linux is an example. Therefore, the Linux kernel serves as a medium layer. The Linux kernel, together with other layers such as applications, library and device drivers could also be reusable.
- The Kernel is a modularized and very large component. Abstracting the kernel in lines of code is not efficient. Procedures seem to be a suitable granularity and abstraction level.
- There is no rooted procedure (a rooted procedure is similar to the main(), which is a root of a C program) for the Linux kernel. Hence, the calling relations among the kernel's procedures are intricate.

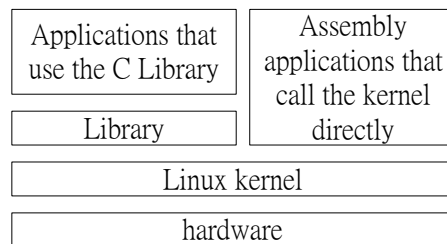


Fig. 1. Linux's layer-based architecture.

Our approach is to use the call graph scheme to represent the three parts of a Linux system, namely, *software applications*, *system libraries*, and *Linux kernel*. Our proposed approach has seven steps included in constructing a Linux application's call graph and a library's call graph:

- Step 1:** Construct an application's call graph from the application source code.
- Step 2:** Construct a Library's call graph from the library's source code.
- Step 3:** Construct a kernel's call graph from the source code.
- Step 4:** Identify the needed hardware devices for an embedded system.
- Step 5:** Combine the application's call graph, library's call graph, and kernel's call graph for the system calls that an application actually needs.

Step 6: Identify which exception handlers the kernel needs.

Step 7: Remove the unused (uncalled) procedures and test the new kernel.

The advantage of this approach is in reusing each layer's asset, and in easily coping with the above issues when customizing the Linux kernel for a specific application. In the remainder of this section, details of each step are illustrated.

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

The first step of our approach is to construct an application's call graph since the source code is available. Most Linux applications are implemented in C or C++. Based on the concept of the call graph, the call structure of this application can be easily constructed.

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 $P \cup s$ is the set of nodes (the nodes represent the procedures of this program), and the set E is defined by the call relations on $(s \cup P) \times P$, i.e., a directed edge $(p_1, p_2) \in E$ exist iff p_1 calls p_2 one or more times. p_1 is said to be the predecessor of p_2 , and p_2 is said to be the successor of p_1 . $Path(px, py)$ denotes that there exists a path connecting px to py . s is the initial node of CG . This node is a call graph entry, and if $p_i \in P$, then there is at least one $path(s, p_i)$, i.e., there are no uncalled procedures in P . If there exists a $Path(p_1, p_2)$ between p_1 and p_2 , then $p_2 \in SUCC(p_1)$.

A call graph represents a program's static structure. Take a C program for example. It can be found that the path is composed of a number of invoked procedures with *main* () as the starting node of this path (Fig. 2 (a)). $SUCC(main)$ should contain the necessary procedures. Hence, if there exists a procedure $P \notin SUCC(main)$, P is the unused procedure of this program. Consequently, P could be removed from this program. This is an interesting question: why does there exist an unnecessary procedure? This situation usually appears due to a programming mistake, especially in a huge program like Linux. Although procedure P has not caused any fault within Linux, we can't be sure that it is safe when porting it to an embedded system. In the case of Fig. 2 (b), two procedures, $e()$ and $f()$, have to be removed.

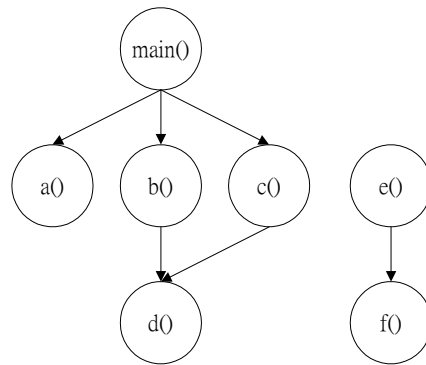
Step 2: Construct a Library's call graph from the library's source code.

Typically, a Linux C program requests an OS's services through library calls (Fig. 3 (a)). In Linux, libraries such as I/O functions, mathematics functions, and string functions, etc., usually occupy a good deal of space. Most embedded Linux companies prefer to develop a new library from scratch. However, from the viewpoint of software reuse, the time spent developing a library could be reduced if library calls are reused in an efficient way.

Therefore, the second step of this approach is to construct a call graph for libraries. This graph also represents a library's calling structure, but it will have no root. In other words, it provides a number of entries for an application written in C (Fig. 3 (b)). The purpose of analyzing a library's calling structure is to find which library calls might be reusable. Of course, we may be able to predict the unnecessary ones from this call graph.

```

main()
{ a(); b(); c(); }
a()
{ ... }
b()
{ d(); }
c()
{ d(); }
d()
{ ... }
e()
{ f(); }
f()
{ ... }
    
```



(a) Application source code.

(b) Application call graph.

Fig. 2. Generating a call graph from the application source code.

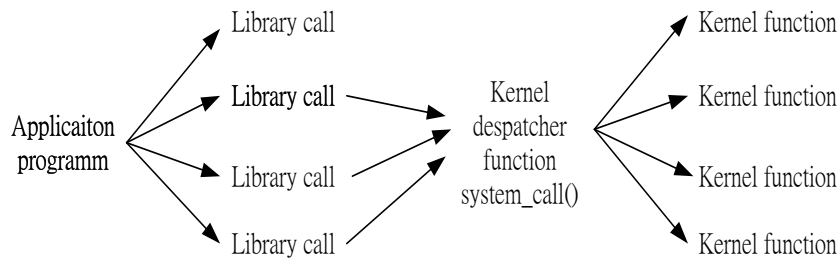


Fig. 3 (a). Call a system service by library call.

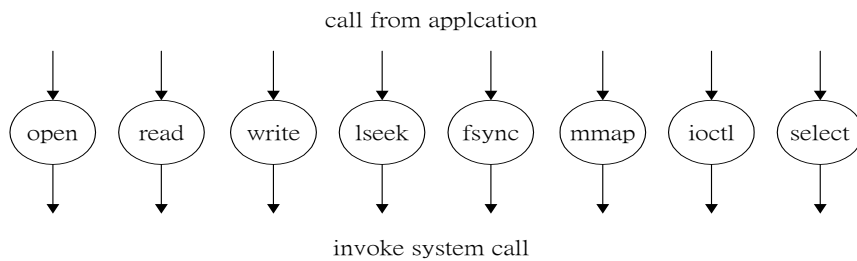


Fig. 3 (b). Simplified library call graph.

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

The next layer below the library is the kernel, the central part of Linux. It controls and coordinates processes’ activities. Because the kernel is an important and big building block, most embedded Linux distributions prefer to reserve the kernel instead of changing it to keep kernel’s correct execution. Therefore, there might be much unnecessary code residing in the kernel.

The third step of this approach is to analyze the kernel's calling structure. Similar to step 2, the kernel's calling structure has several entries for its underlying layer. Fig. 4 depicts the relation among applications, library, and kernel. Most modern operating systems are interrupt-driven systems. Asynchronous requests ask for the system services through interrupts or exceptions. Whenever an interrupt or exception occurs, the kernel will initiate the corresponding handlers to service it. Therefore, to abstract the Linux kernel, we first must understand when the kernel will be activated and executed. A list of occasions is as follows:

- Booting the Linux system
- Invocation of system calls from an application
- Occurrence of an exception. Whenever this situation occurs, a user process will be suspended until kernel handles the exception.
- Occurrence of an interrupt. Any time a hardware device generates an interrupt signal, the kernel will be activated to handle it.

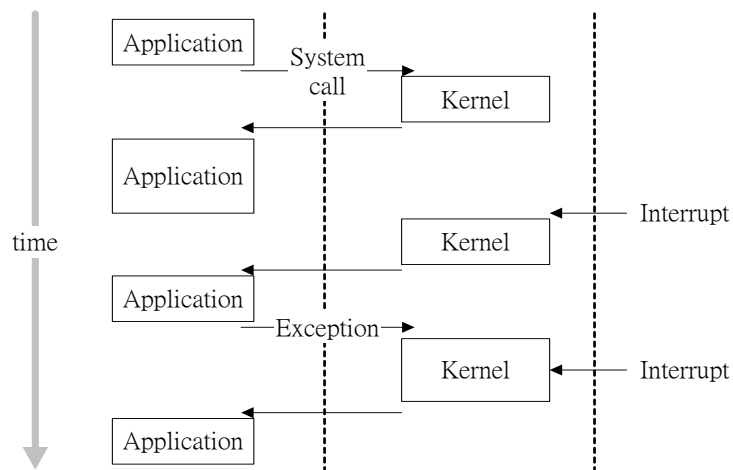


Fig. 4. Interleaving of kernel control path.

The second and the third conditions are application related, and the fourth condition is hardware dependent. We can identify unnecessary kernel procedures by observing the kernel's call graph together with the above-mentioned conditions. We can then combine an application's call graph, the library's call graph, and the kernel's call graph. To generate a SUCC(main). Any procedures not included in this set may be removed.

Step 4: Identify the needed hardware devices for an embedded system.

Linux supports many hardware devices, such as disk, keyboard, and mouse. However, an embedded system is a simplified platform, and devices like a keyboard or a mouse may not be needed in an embedded system, like a PDA. Therefore, when porting

Linux into an embedded system, a designer will encounter issues such as too many unused device drivers and associated program code in Linux. Fortunately, many modern peripherals are plug-and-play devices. Linux allows an experienced engineer to decide which drivers should or should not be loaded. However, much peripheral manipulation data and code still reside in the kernel. These components might unavoidably occupy some of the embedded system's resources. Therefore, the fourth step is to find and remove such unused components.

In step 4, we focus on two situations concerning the interactions between kernel and hardware:

- **Device initialization:** Usually, Linux will test and then initialize all of the possible devices since the device configuration may change between system bootings. However, device configuration is usually fix and thus only the initialization procedure for predefined devices is required in an embedded system. For example, a network camera doesn't contain a screen monitor, mouse or keyboard. Therefore, the unnecessary initialization code for unavailable devices should be removed.
- **Drivers for unavailable devices:** Another similar consideration is to remove the drivers for unavailable devices. Many embedded systems engineers prefer to remove all drivers and develop in-house ones.

Fortunately, Linux provides functionality, "*make config*", to add/remove hardware validation code. The complexity of this situation, however, depends on engineer's experience. For the second condition, Linux kernel provides diverse interfaces located in the top layer for an application to access devices including *lseek*, *read*, *write*, *readdir*, *select*, *ioctl*, *mmap*, *open*, *release*, *fsync*, *fasync*, *check_media_change* and *revalidate*. We want to find the unnecessary interfaces and remove them from kernel.

Step 5: Combine application's call graph, library's call graph, and kernel's call graph for the system calls that an application actually needs.

From step 5 to step 7, we want to find a set of SUCC(main) which describes a number of procedures that are capable of being reused for a specific application. As discussed above, a library's call graph lacks a root. Therefore, we incorporate the application's call graph into the library's call graph. Main() is the root of the application's call graph, representing the unique entry of this combined call graph. Moreover, according to the SUCC(main), the unused library's calls are removed. Consequently, a customized library is obtained.

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

Step 6: Identify which exception handlers the kernel needs.

In step 6, we remove the unused code that handles exceptions. As we know, some exception handlers are actually not needed. There are 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 the kernel.

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

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

3. A CASE STUDY: AN ACDC CD PLAYER

In this section, we apply the call graph approach to a case study called ACDC player, which ACDC is a media application running on Linux. In order to demonstrate our proposed approach, we adapt ACDC to an embedded application.

In section 3.1, we describe how to construct the call graph for ACDC. In section 3.2, some experimental measurements are shown to demonstrate the feasibility of our approach.

3.1 Customization of ACDC CD Player

Our approach can be summarized in three phases:

- Construct call graphs for application, library and kernel.
- Find the unused code, procedures, and drivers.
- Remove these and test the new kernel.

To reuse and customize ACDC, it is necessary to first construct the related call graphs. Then we combine the three call graphs and go on to obtain SUCC(main). In this case, our experimental environment is as follows:

- Linux distribution: Slackware 3.0
- Linux Kernel version: 1.2.3
- C Library version: libc 4.6.27
- Target application: text mode ACDC CD player

Linux kernel version 1.2.3 has been adopted for academic research due to its simplicity. Although version 1.2.3 doesn't support overly complex functionalities that are seldom used in embedded systems, it is indeed easier to analyze this kernel's structure. ACDC is a text mode application running on this kernel. A GUI for a CD player is not necessary. On the contrary, a text mode ACDC is adequate and can simplify our demonstration.

First, we construct ACDC's call graph to find the necessary procedures. Fig. 5 depicts the simplified call graph of ACDC. Fig. 5 shows a customized result, because many unused procedures have been eliminated. Since ACDC player can be a handles system, a display and keyboard are not necessary. We can thus eliminate its user interface.

Several procedures listed in Fig. 5 are actually provided by *libc*. These procedures can be regarded as the entries that a Linux application requests from the next layer's services. The next step, therefore, is to construct *libc*'s call graph. By examining this call graph, unused library calls can be found and removed. The gray box in Fig. 6 depicts the corresponding *libc*'s call graph.

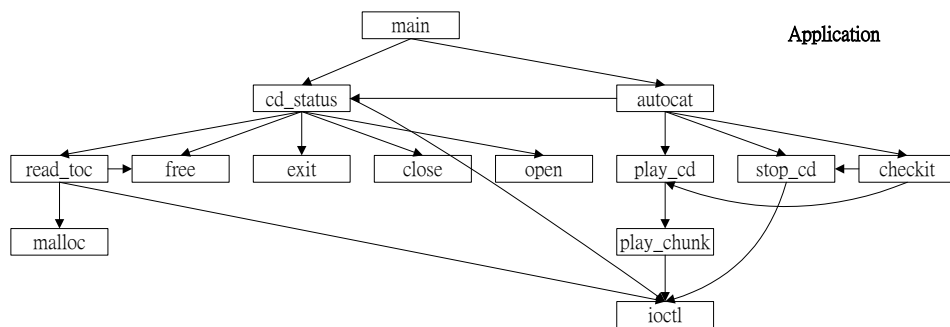


Fig. 5. ACDC's call graph.

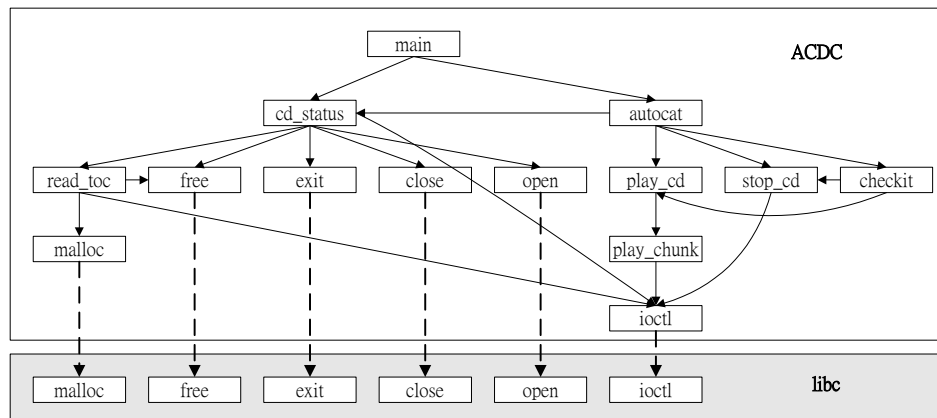


Fig. 6. The combination of ACDC's call graph and *libc*'s call graph.

Six *libc*'s calls, *malloc*, *free*, *exit*, *close*, *open*, and *ioctl*, are channels between ACDC and *libc*. This means that these six *libc*'s calls should be included in *SUCC*(main). Therefore, we can simply remove the rest of the *libc* calls.

The next step is to combine Fig. 6 with the kernel's call graph as shown in Fig. 7. Fig. 7 shows that the kernel provides five system calls, *mmap*, *exit*, *close*, *open*, and *ioctl*.

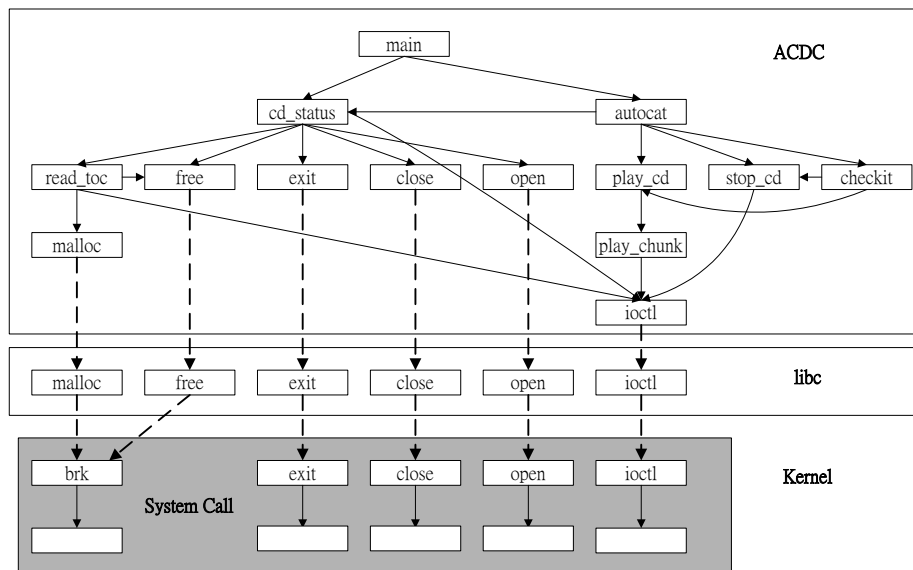


Fig. 7. The combined call graph.

Each of these will call its successors with respect to its own sequence. Finally, through the cooperation among the system calls, the kernel can successfully serve its requests from the upper layer. The system calls that are not included in $SUCC(main)$ will thus be removed.

3.2 Evaluation Results

The Linux kernel can be decomposed into several parts, each of which is stored in individual directory. There are ten major source subsystems in the Linux kernel source tree:

- The *arch* directory contains the kernel source code that is specific to particular hardware architecture. *head.o* contains hardware dependent Linux kernel booting code. *kernel.o* contains architecture dependent code in the Linux kernel. *mm.o* contains the architecture dependent memory management code;
- The *init* directory contains the architecture independent Linux kernel boot and initialization code.
- The *kernel* directory contains the main architecture independent functionalities provided by the Linux kernel.
- The *mm* directory contains the architecture independent Linux kernel memory management code.
- The *fs* directory contains various kinds of file systems supported by the Linux kernel. *fs.o* contains virtual file system code. *filesystem.a* contains specific file system codes.
- The *net* directory contains the networking routines, such as sockets, IPX and TCP/IP protocols.

- The *ipc* directory contains the inter-process communications code.
- The *drivers* directory contains block, char, net, scsi and sound device driver code.
- The *lib* directory contains the library code.
- The *include* directory contains most of the Linux kernel's include files.

Table 1 column (A) lists the program size (in bytes) for each Linux kernel object file and library file after issuing a “*make config*” command. Column (B) lists the program size (in bytes) of each Linux kernel object file and library files after the proposed elimination approach. Column (C) lists the reduction size in the column (B) in percentage. Because the code for most of the system calls are stored in the kernel directory, by using our proposed customization approach, about 45 percent of the code in *kernel.o* will be eliminated.

Table 1. Measurements of case 1.

Structure of Linux kernel's codes	Linux kernel code size after “ <i>make config</i> ” (A)	Linux kernel code size after using call graph elimination approach (B)	Reduced Percentage (C)
arch/i386/kernel/head.o	60,617	60,617	0 %
arch/i386/kernel/kernel.o	41,227	34,601	16.1 %
arch/i386/mm/mm.o	3,593	2,882	19.8 %
init/main.o	7,732	6,134	20.7 %
init/version.o	639	639	0 %
Kernel/kernel.o	61,469	34,021	44.7 %
mm/mm.o	34,351	29,342	14.6 %
fs/fs.o	79,403	62,561	21.2 %
net/net.o	15,075	1,635	89.2 %
ipc/ipc.o	226	226	0 %
fs/filesystem.a	73,312	61,246	16.5 %
drivers/block.a	50,120	50,120	0 %
drivers/char.a	136,648	125,910	7.9 %
drivers/net.a	6,516	4,160	36.2 %
lib/lib.a	7,084	5,828	17.7 %
net/network.a	6,642	360	94.8 %
Compressed kernel size	183,300	150,532	17.9 %
Uncompressed kernel size	466,671	387,331	17.0 %

Column (D) repeats the same data as Table 1 column (A). Table 2 column (E) shows the result after using the call graph elimination approach and manually processing to eliminate the code for *printk()*, *panic()* and *char.a*. We can treat these functions as dead functions (i.e., dead code). The result shows that the Linux kernel size can be reduced by 43.1 percent.

Table 2. Measurements of case 2.

Structure of Linux kernel's codes	Linux kernel code size after "make config" (D)	Linux kernel code size after using call graph elimination approach (E)	Reduced Percentage (F)
arch/i386/kernel/head.o	60,617	60,573	0 %
arch/i386/kernel/kernel.o	41,227	32,219	21.8 %
arch/i386/mm/mm.o	3,593	2,000	44.3 %
init/main.o	7,732	5,104	34.0 %
init/version.o	639	639	0 %
kernel/kernel.o	61,469	32,091	47.8 %
mm/mm.o	34,351	23,031	33.0 %
fs/fs.o	79,403	57,530	27.5 %
net/net.o	15,075	1,535	89.8 %
ipc/ipc.o	226	226	0 %
fs/filesystem.a	73,312	55,122	24.8 %
drivers/block.a	50,120	41,076	18.0 %
drivers/char.a	136,648	0	100.0 %
drivers/net.a	6,516	4,160	36.1 %
lib/lib.a	7,084	5,828	17.7 %
net/network.a	6,642	360	94.6 %
compressed kernel size	183,300	97,284	46.9 %
uncompressed kernel size	466,671	274,660	41.1 %

4. CONCLUSIONS AND FUTURE WORK

In this paper, a call graph representation is adopted to depict the Linux kernel's calling structure. The call graph is an adaptive structure capable of being modified to meet different application-specific domains. This property is beneficial for embedded operating system design. Our approach, unlike typical methodologies that customize a Linux kernel, is to employ call graphs to represent the structure of the kernel in achieving the goals of efficacy, flexibility and low cost.

An experiment using an ACDC player is demonstrated. The results show that our approach can remove about 17 % of a kernel's footprint with respect to unreachable code and causes no side effects. In addition, 41.1% of the kernel's footprint can be removed, both unreachable code and dead code, without causing any errors. Although the proposed procedure call level approach could significantly reduce the size of Linux kernel source code, we can not claim that the customized kernel is minimal. An instruction-level minimization scheme might be needed to get greater reduction. The paper aimed to first provide our experience of a systematic kernel customization using call graphs. Several future works based on this research could be focused, for example, on theoretically proving the correctness of the customized kernel with the proposed scheme; proposing an instruction-level scheme for further reducing the kernel size; customizing the kernel with con-

siderations of advanced system features like multiprocessing support; developing a visualized development toolkit to remove redundant code from the Linux kernel in a more accurate and efficient way; and so on.

REFERENCES

1. TiVo digital recorder, <http://www.tivo.com>.
2. RIO MP3 player, <http://www.riohome.com>.
3. Axis Network Camera, <http://www.axis.com>.
4. G. Snelting, "Reengineering of configurations based on mathematical concept analysis," *ACM Transactions on Software Engineering and Methodology*, Vol. 5, 1996, pp. 146-189.
5. D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, Vol. 26, 1994, pp. 345-420.
6. S. K. Debray and W. Evans, "Compiler techniques for code compaction," *ACM Transactions on Programming Languages and Systems*, Vol. 22, 2000, pp. 378-415.
7. uCLinux, <http://www.uclinux.com>.
8. D. Callahan, A. Carle, M. W. Hall, and K. Kernedy, "Constructing the procedure call multigraph," *IEEE Transactions on Software Engineering*, Vol. 16, 1990, pp. 483-487.
9. M. S. Hecht, *Flow Analysis of Computer Programs*, North-Holland, New York, 1977.
10. B. Ryder, "Constructing the call graph of a program," *IEEE Transactions on Software Engineering*, Vol. 5, 1979, pp. 216-225.
11. A. Cimitile and G. Visaggio, "Software salvaging and the call dominance tree," *The Journal of Systems and Software*, Vol. 28, 1995, pp. 117-127.
12. E. Burd and M. Munro, "A method for the identification of reusable units through the reengineering of legacy code," *The Journal of Systems and Software*, Vol. 44, 1998, pp. 121-134.
13. J. P. Banning, "An efficient way to find the side effects of procedure calls and the aliases of variables," in *Proceedings of ACM Symposium on Principles of Programming Languages*, 1979, pp. 29-41.
14. ACDA, http://www.hitsquad.com/smm/linux/CD_PLAYERS/.
15. C. T. Lee, Z. W. Hong, and J. M. Lin, "Linux kernel customization for embedded system by using call graph approach," *Asia and South Pacific Design Automation Conference*, 2003, pp. 689-692.
16. C. T. Lee, Z. W. Hong, and J. M. Lin, "An application-oriented Linux kernel customization for embedded systems," *National Computer Symposium*, 2003.



Chi-Tai Lee (李啓泰) received the B.S. degree in Information Engineering and Computer Science from the Feng Chia University, Taiwan, in 1995. He is currently a Ph.D. candidate in the Department of Information Engineering and Computer Science, Feng Chia University, Taiwan. His research interests include operating systems, embedded systems, and software engineering. He has been involved in projects of porting the Mach 3.0 microkernel operating system to massively parallel processing (MPP) evaluation system, reengineering commercial off-the-shelf (COTS) software, and embedded systems.



Jim-Min Lin (林志敏) was born on March 5, 1963 in Taipei, Taiwan, R.O.C. He received the B.S. degree in Engineering Science and the M.S. and the Ph.D. degrees in Electrical Engineering, all from National Cheng Kung University, Tainan, Taiwan, R.O.C., in 1985, 1987, and 1992 respectively. Since February 1993, he has been an Associate Professor at the Department of Information Engineering and Computer Science, Feng Chia University, Taichung, Taiwan, R.O.C. During August 2001 and July 2003, he was on leave and served as an Associated Professor and Chairman of the Department of Information Management, Taichung Healthcare and Management University, Taichung, Taiwan. His research interests include operating systems, software integration/reuse, embedded systems, and software agent technology.



Zeng-Wei Hong (洪振偉) received his M.S. degree in Information Engineering and Computer Science from Feng Chia University, Taiwan, in 2001. He is currently a Ph.D. candidate in the Department of Information Engineering and Computer Science, Feng Chia University, Taiwan. His research interests include software engineering, object-orientation technology, design pattern, and software architecture. He has been involved in projects of reengineering commercial off-the-shelf software (COTS), distributed software integration and agent-oriented systems.



Wei-Tsong Lee (李維聰) receive the B.S.E.E. and the M.S. and Ph.D. degrees in Computer Science all from National Cheng Kung University, Taiwan, R.O.C., in 1984, 1986 and 1995. He is currently an Associate Professor in Department of Electrical Engineering of Tamkang University, Taiwan, R.O.C. His current interests include high speed networks, cable modems and stochastic ordering.