Standard Operating Procedures for Embedded Linux Systems

Follow these procedures for the smoothest path to great embedded Linux.

CHI-HUNG CHOU, TSUNG-HSIEN YANG, SHIH-CHIANG TSAO AND YING-DAR LIN

Procedures for developing embedded systems are very complicated. New engineers typically take a long time to become familiar with these procedures. Therefore, we have developed a standard operating procedure (SOP) to save the costs of constructing an embedded system and reduce the complexity. The SOP includes five standard procedures for building a Linux-based embedded system, as shown in Figure 1. You can follow the procedures discussed in this article for building a prototypal system. Also, we introduce ten useful methods for downsizing your system. Finally, we show the effect of these methods on downsizing your embedded system—a content-aware network security gateway.

Five Standard Procedures

To build an embedded system, the first step is to select a target platform. The platform involves both hardware and software. The hardware platform includes the processor, bus and I/O; the software platform includes the bootloader, kernel and root filesystem. You must select each item in the target platform carefully to ensure that the hardware and software work together. For instance, bootloaders relate directly to the hardware. If the selected bootloader does not support your hardware platform, the whole embedded system cannot power on. Moreover, an operating system that requires MMU may fail to collaborate with MMU-supported processors.

Second, in addition to the target platform, a development platform also is necessary. You cannot compile embedded software programs on the target platform, because the target platform often has a small RAM and slow CPU to minimize cost and power consumption. Therefore, you need to prepare a development platform with a fast CPU and large RAM to compile these programs. Besides, because the two platforms have different hardware architectures, a cross-compiler environment is necessary. Buildroot is such a package to offer this environment. It has a friendly user interface to assist in choosing the hardware platform and the required software package. By using Buildroot, you can generate a cross-compilation toolchain and a root filesystem easily with built-in application packages for your embedded system.

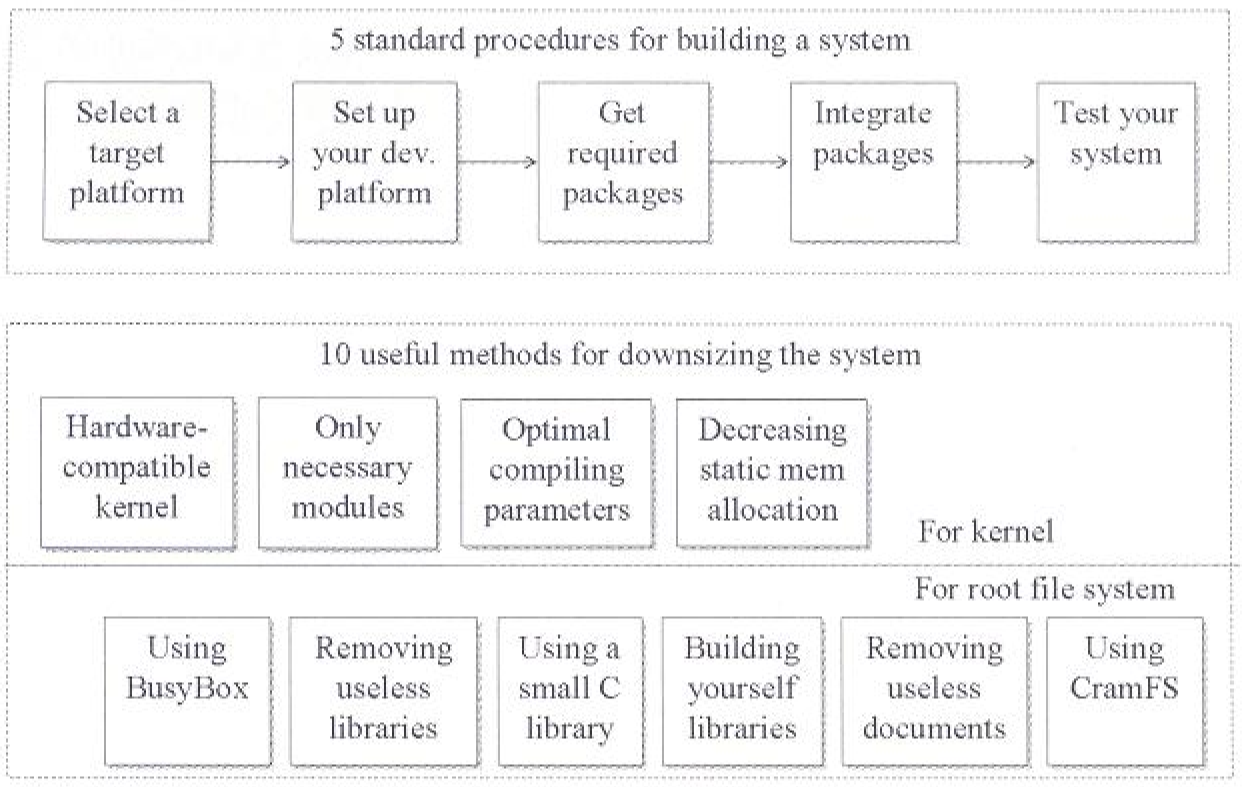

5 standard procedures for building a system
Select a target platform
Set up your dev. platform
Get required packages
Integrate packages
Test your system
10 useful methods for downsizing the system
Hardware-compatible kernel
Only necessary modules
Optimal compiling parameters
Decreasing static mem allocation
For kernel
For root file system
Using BusyBox
Removing useless libraries
Using a small C library
Building yourself libraries
Removing useless documents
Using CramFS

Figure 1. Procedures and Methods for Building and Downsizing Your Embedded Systems

After setting up the environment, the next step is identifying the packages required by your system. You can accomplish this by selecting the built-in packages directly from the menuconfig of Buildroot, or you can download them from the Internet. In fact, Buildroot provides a list of useful packages, such as iproute2, freeswan and squid. Buildroot also ensures that these packages can link successfully with uClibc, a C library with a smaller size than Glibc. If you cannot find the suitable packages, you will have to modify existing packages or write new ones.

Having obtained the required packages, the next step is integrating them into the embedded system. Integrating here means using a cross-compiler to compile the source code into forms that can be executed in the target platform, and then adding them into the root filesystem. You can add packages into the root filesystem through Buildroot in any of three ways, as shown in Figure 2. Method 1 is to select them directly from the options in Buildroot. If the packages are not available in Buildroot, you may need to write a makefile for the package to indicate how to download, configure, compile and install the package. Also, you need to modify

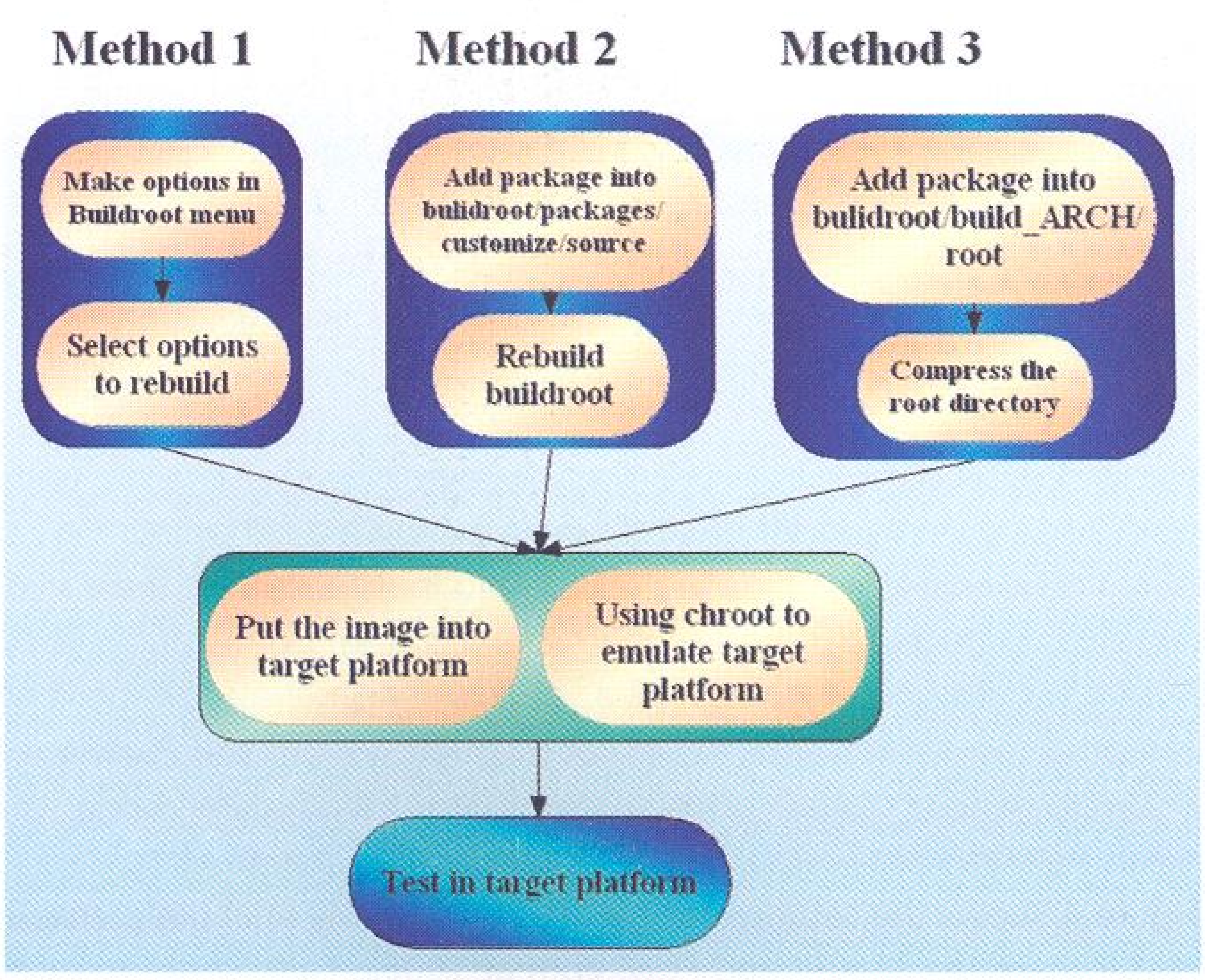

Method 1
Method 2
Method 3
Make options in Buildroot menu
Select options to rebuild
Add package into bulidroot/packages/customize/source
Rebuild buildroot
Add package into bulidroot/build_ARCH/root
Compress the root directory
Put the image into target platform
Using chroot to emulate target platform
Test in target platform

Figure 2. Three Methods for Adding Packages into Your Root Filesystem

the config.in file of Buildroot to display the option of the package in the configuration menu. However, if you would not like to write these configuration files, you can use Method 2. In Method 2, you simply place the compiled packages in the directory named customize, and then Buildroot copies these compiled packages into the root filesystem during the building procedures, according to the rules given in customize.mk. However, if you simply want to verify the functionality of a single package, you don't need to rebuild the whole image. The steps in Method 3 are to mount the root filesystem on any one directory and then copy the compiled packages into the directory. Finally, unmount the directory, and you will get an updated root image. However, Method 3 may fail if the free space in the mounted filesystem is not enough for the new packages. In that case, you can adjust the parameters given in ext2root.mk to reserve more free space in your root filesystem during the period of system development.

Finally, Figure 2 depicts two ways to test and verify whether the functions of a package are normal. The basic way is to download the root filesystem into the target platform and execute the package directly. However, doing this usually takes a long time. Another way is to boot the root filesystem in a virtual machine, such as QEMU and VMware. Using a virtual machine to examine a compiled image is fast and convenient, but a virtual machine may not simulate some characteristics, such as hardware interrupts. Hardware interrupts involve quick reaction behavior, so they cannot be implemented by virtual machines easily. Finally, if the target platform has the same CPU architecture as your development platform, you can use chroot to replace your local system with the target root filesystem.

By following the five procedures outlined, you can build the root filesystem for your embedded system. However, there is still one problem that may be troubling you—how to downsize your embedded systems or how to use less Flash RAM to store the kernel and root filesystem. Requiring less RAM means that you can cut the cost of your embedded system.

Ten Downsizing Methods

The organization of the downsizing issue is displayed in the bottom of Figure 1. We divide the methods into two parts, because the software platform of an embedded system typically consists of a kernel and root filesystem. The first part is how to get a small kernel, and the second part is how to downsize each component in the root filesystem, including libraries and shells. The second part also discusses how to compress the whole root filesystem. We describe all methods in detail below, along with experimental results. After explaining all methods, we show the effect of these methods on our laboratory embedded system, called the Wall system. Table 1 presents the specification for the Wall system. The system is a network security gateway that provides application-layer content filters, such as antispam and antivirus.

Methods for the Linux Kernel

Selecting an appropriate kernel is the first step in downsizing the kernel. If you choose an inappropriate kernel, the system may be not only large but also unable to use processor power effectively. For example, a standard Linux kernel on a hardware platform without MMU cannot work normally. Such a hardware platform requires a specific MMU-less kernel, such as uClinux. Most people use the standard Linux kernel and attempt to trim its size.

The next step is to include only the necessary modules in the standard Linux kernel by a correct configuration. In fact, the default configuration of a Linux kernel includes many unused modules, which causes you to have a big kernel. Figure 3(a) shows the experimental results on the downsizing effect of the correct configuration. In this case, a system supporting TCP/IP has a kernel image that is only 59.84% of the size of a system supporting all network protocols.

To downsize the kernel, the third step is to use the optimization parameters when compiling the kernel. Using parameters -O1, -O2 or -O3 can improve performance, and using -Os can reduce size. However, optimizing for both performance and size simultaneously is not possible. Therefore, we generally select -O2 to achieve a balance between size and performance. As shown in Figure 3(b), the -Os parameter reduces the size of the kernel image by 22.82% as compared with -O3, but it causes worse performance.

Besides including only the necessary modules and compiling the kernel with the optimal parameters, to downsize the kernel further, you can

Table 1. Specification of Wall

	FUNCTIONS	PACKAGES
Kernel	X86, MMU, QoS, Ethernet, Wireless	Linux 2.6.6; 1,302,362 Bytes
Connection	LAN, DMZ, WAN, DHCP, DNS relay, Dynamic DNS, Link load balance, Bridge mode	ppp-2.4.1, rp-pppoe-3.5
Security	IPSec, PPTP, L2TP, SSL-VPN	freeswan2.06, 12tpd-0.69
Firewall	NAT, firewall, UPNP, traffic profiling, APP firewall	iptables-1.2.9, hotplug, iproute2
Mail	Antispam, antivirus, POP3 proxy	p3scan
Web	Transparent proxy, URL, URL keyword, content keyword	p3scan
IM	MSN log	Development based on L7Filter
BW Control	TC	TC
Management	Web, SSL, FTP, log rotation	thttpd-2.21b, Openssl-0.9.7d, putre-ftpd-1.0.17a, cron
Platform	i386, IXP (simple version)	

decrease the size of the static buffer and array allocated in the kernel, because the kernel typically declares a large buffer and array for standard PCs. To find out which buffer or array occupies large memory space, you can use the command nm. This command can list the allocated size of each variable in an object file. With that information, you can browse the corresponding source code of the object file and alter the initial size of the buffer or array. Another approach for shrinking the buffer size is to modify the options in the menuconfig of the kernel to decrease the maximum number of supported peripherals, as shown in Figure 3(c) and (d).

Methods for the Root Filesystem

As shown in Figure 1, we identify six methods for downsizing the root filesystem. First, you can adopt a tool called BusyBox, which provides a fairly complete environment for any small or embedded system. BusyBox combines tiny versions of many common UNIX utilities into a single small executable file, and it is highly modular, allowing commands to be included or excluded at compile time. The space used for BusyBox is 7.04% of that of the original tool, as demonstrated in Figure 4.

Next, we introduce three methods for removing unused libraries or downsizing required libraries. First, you can use the command ldd to identify the required shared libraries for each program, and then with this information, you can remove the unused libraries. Notably, if a shared library is not used by programs, you additionally should check whether it is used by other shared libraries. Figure 4 shows that removing redundant libraries reduces the root filesystem to 6.55% of its original size. Second, you can replace the standard C library with a small C library, such as uClibc, Newlib or diet libc. Such libraries remove the unused functions, so their size is smaller than Glibc, as shown in Table 2. This table presents the differences in functionality between the four libraries. Third, you can use a library optimizer tool named Libopt to rebuild the libraries that include the only necessary functions for the executable programs and shared libraries found in the root filesystem. This tool utilizes objdump and nm to gather information about library object files, shared libraries and executable programs.

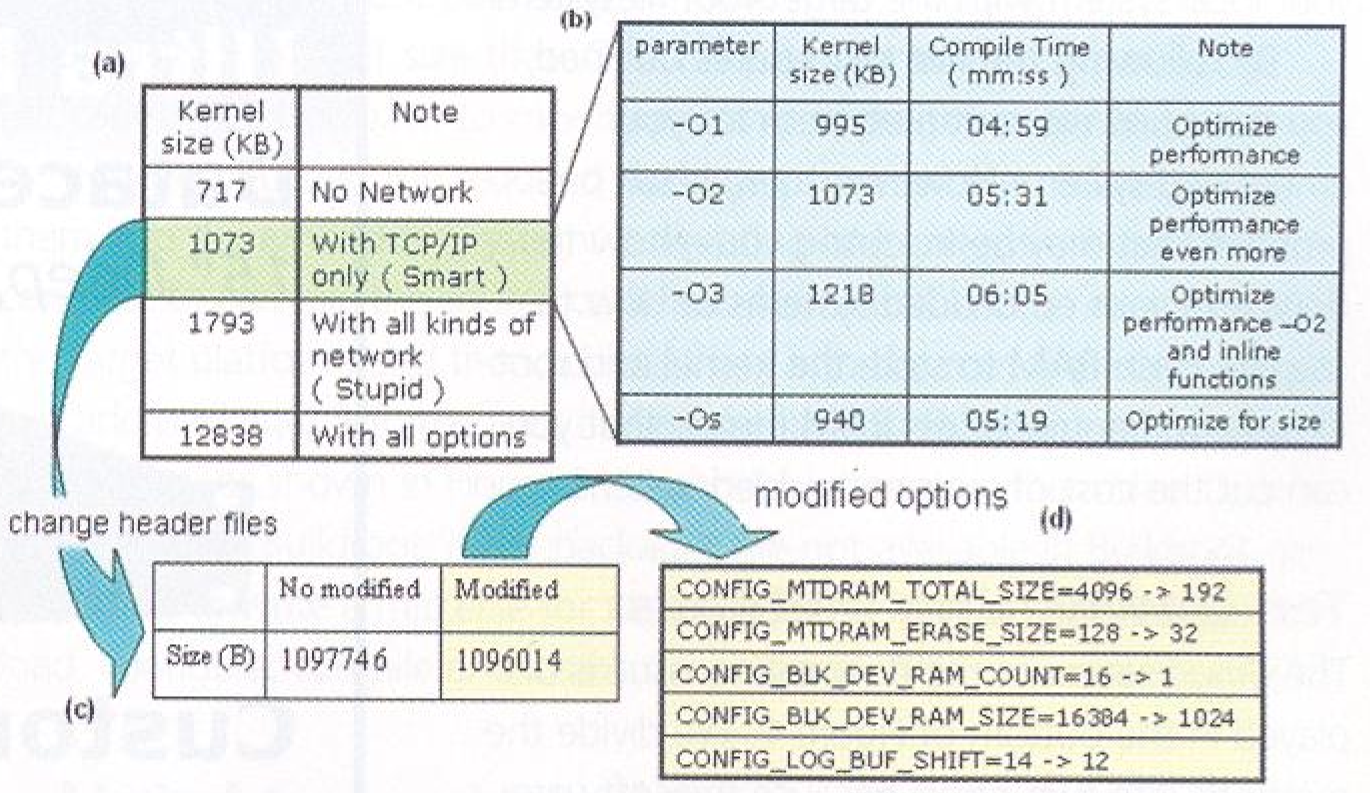
The fifth method for downsizing the root filesystem is to remove

Table 2. Comparison between Different C Libraries

	GNU C Library	uClibc	diet libc	Newlib
Size	Largest	Small	Smallest	Small
Compatibility	Good	Good	Bad	Normal
Speed	Fastest	Fast	Fast	Fast
Portability	Yes	Yes	Yes	Yes
MMU-less supporting	No	Yes	Yes	Yes
Licensing	LGPL	LGPL	LGPL	BSD, GPL
Setting		menuconfig	only make	./configure
Note	Standard C library	Needs cross-compiler toolchain	Often linked as static library	Managed by Red Hat

unnecessary documents. You can eliminate some directories, such as /home, /mnt, /opt, /root, /boot and /proc, if unused. You also can remove the man, info, include and example directories to reduce the size when additionally integrating a package into the root filesystem. In general, an embedded system executes only specific programs, so users can operate it easily without the help documents or examples in these directories.

The final method is to avoid uncompressing the whole root filesystem into SDRAM. The root filesystem is compressed to save the stored space, for example, Flash RAM. However, after the filesystem is uncompressed into SDRAM, the Flash memory allocated for the filesystem is no longer necessary. For instance, if the compressed size of the root



Kernel size (KB)	Note
717	No Network
1073	With TCP/IP only (Smart)
1793	With all kinds of network (Stupid)
12838	With all options

parameter	Kernel size (KB)	Compile Time (mm:ss)	Note
-O1	995	04:59	Optimize performance
-O2	1073	05:31	Optimize performance even more
-O3	1218	06:05	Optimize performance –O2 and inline functions
-Os	940	05:19	Optimize for size

	No modified	Modified
Size (B)	1097746	1096014

```
CONFIG_MTDRAM_TOTAL_SIZE=4096 -> 192
CONFIG_MTDRAM_ERASE_SIZE=128 -> 32
CONFIG_BLK_DEV_RAM_COUNT=16 -> 1
CONFIG_BLK_DEV_RAM_SIZE=16384 -> 1024
CONFIG_LOG_BUF_SHIFT=14 -> 12
```

Figure 3. Effects of the Downsizing Methods on the Kernel

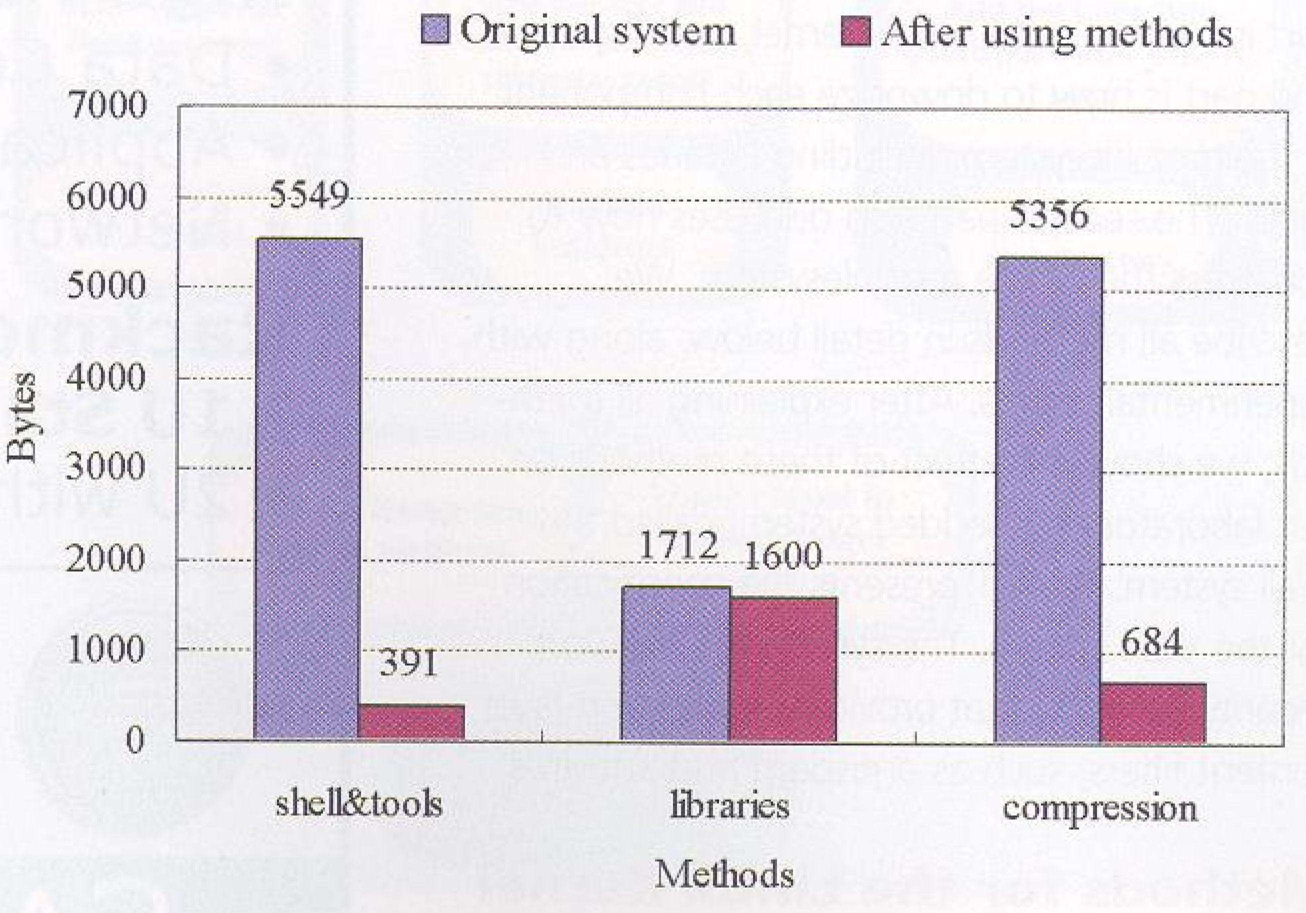
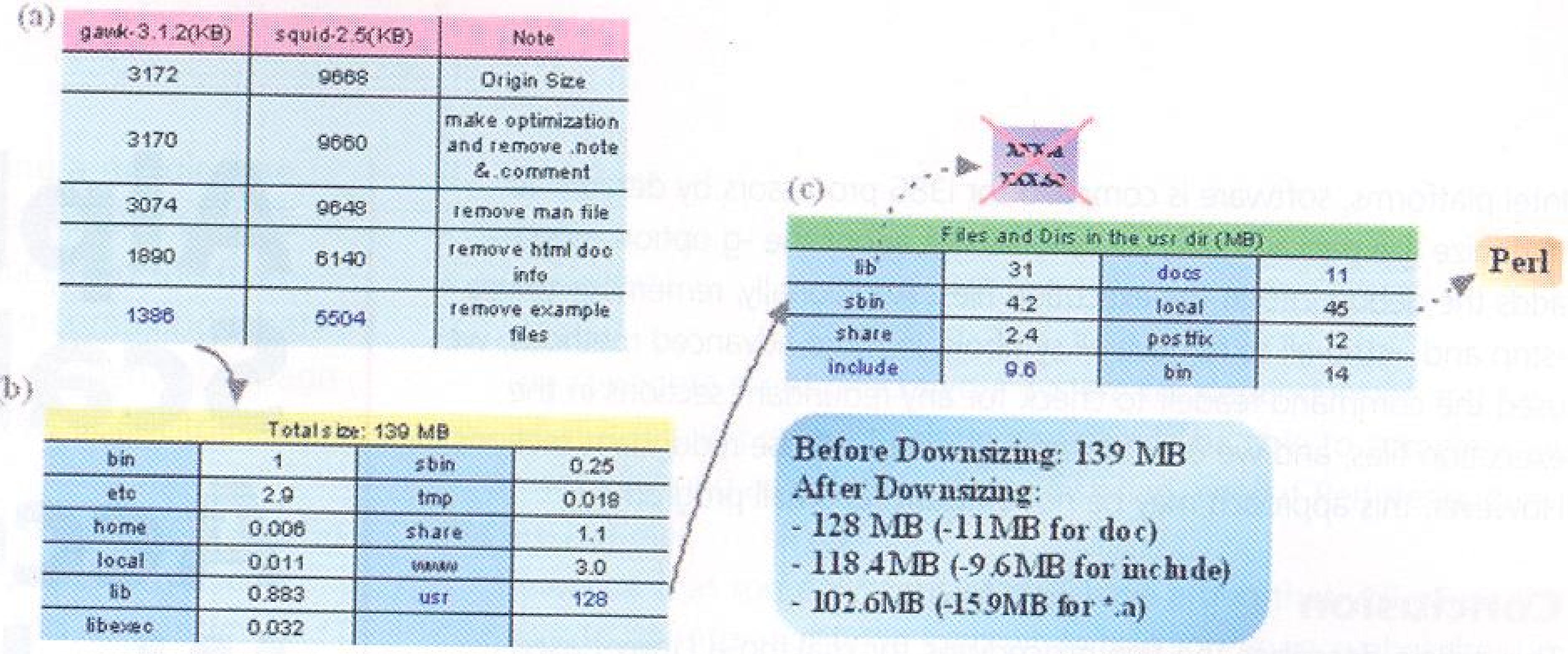


Figure 4. Effects of Downsizing Methods on the Root Filesystem

filesystem is 4MB and its compression rate is 50%, the system occupies 4MB of Flash memory and 8MB of SDRAM. Therefore, the system wastes much memory storage, because of the duplicate data. For this problem, you can use CRAMFS. CRAMFS is a read-only filesystem, designed for simplicity and space efficiency. You do not need to uncompress a CRAMFS image before mounting it. A CRAMFS image is zlib-compressed, one page at a time to enable random read access. The metadata is not compressed, but is expressed in a terse representation that is more space-efficient than in traditional filesystems, such as ext2 or FAT. However, due to the read-only property of compressed files, random write access is hard to implement for them. As shown in Figure 4, CRAMFS compresses the filesystem to 12.77% of its original size.



(a)

gawk-3.1.2(KB)	squid-2.5(KB)	Note
3172	9668	Origin Size
3170	9660	make optimization and remove .note &.comment
3074	9648	remove man file
1890	6140	remove html doc info
1386	5504	remove example files

(b)

Total size: 139 MB			
bin	1	sbin	0.25
etc	2.9	tmp	0.018
home	0.006	share	1.1
local	0.011	www	3.0
lib	0.883	usr	128
libexec	0.032		

(c)

Files and Dirs in the usr dir (MB)			
lib	31	docs	11
sbin	4.2	local	45
share	2.4	postfix	12
include	9.6	bin	14

Before Downsizing: 139 MB
After Downsizing:
- 128 MB (-11MB for doc)
- 118.4MB (-9.6MB for include)
- 102.6MB (-15.9MB for *.a)

Figure 5. Downsizing Results on the Root Filesystem of the Wall Project

Now that we've covered the six methods, let's move on to the effect of these methods on the Wall Project, as shown in Figure 5. First, we used BusyBox to substitute for the multiple utility programs used in the original shell. Then, we compiled all the required packages with the parameters --strip-unneeded and -O2. Next, we used the commands strip and objcopy to remove the unnecessary contents of packages. Finally, we deleted unnecessary directories, such as man, info and example. Figure 5(a) illustrates the result of these processes. However, the size of Wall was still 139MB. Hence, we had to view the contents of /usr indepth, as shown in Figure 5(b) and (c). In the Wall Project, removing unneeded documents and files saved 20.6MB of space. About 15.9MB of space then can be saved by eliminating unused libraries. However, as you can see, Perl occupied much space in our system. Other methods may exist to solve this problem, but it is sufficient to consider only what we have done above.

We found that the optimization of package size is also useful for downsizing when integrating a new package into the root filesystem. Actually, most programs and libraries are compiled at optimizing level 2 by default (gcc options -g and -O2) and are compiled for a specific CPU. On

Intel platforms, software is compiled for i386 processors by default. To minimize the package size, you should not adopt the -g option, which adds the debug info in the execution files. Additionally, remember to use -strip and --strip-all to remove all symbols. In more-advanced methods, we used the command readelf to check for any redundant sections in the execution files, and we used objcopy to remove those redundant sections. However, this approach may be not efficient for small programs.

Conclusion

This article describes the five procedures for making a Linux-based embedded system and describes ten methods for downsizing the kernel and the root filesystem. After we used these methods, our Wall Project was downsized by 26.18%. The experiment's results reveal that the two most efficient methods are giving correct kernel compilation parameters and using simplified tools and libraries in the root filesystem. Hopefully, this article helps you understand the procedures and problems when building a Linux-based embedded system.■

Chi-Hung Chou is currently working on his Masters' degree in Computer Science at National Chiao Tung University. His research interests include mesh network and embedded systems. He can be reached via e-mail at payton345.cs95g@nctu.edu.tw.

Tsung-Hsien Yang is currently working on his Masters' degree in Computer Science at National Chiao Tung University. His research interests include automatic block module tests and embedded systems. He can be reached via e-mail at thyang.cs95g@nctu.edu.tw.

Shih-Chiang Tsao is a PhD candidate in Computer Science at National Chiao Tung University and has been advised by Dr Ying-Dar Lin since 2003. His research interests include TCP-friendly congestion control algorithms, fair-queuing algorithms and Web QoS. He can be reached via e-mail at weafon@cs.nctu.edu.tw or through his Web site (www.cs.nctu.edu.tw/~weafon).

Ying-Dar Lin received a PhD in Computer Science from the University of California, Los Angeles (UCLA) in 1993. He has been a professor of Computer Science at National Chiao Tung University since 1999. He also is the founder and director of the Network Benchmarking Lab (NBL), which reviews the functionality, performance, conformance and interoperability of networking products, ranging from switch, router and WLAN to network and content security and VoIP. His research interests include design, analysis, implementation and benchmarking of network protocols and algorithms, wire-speed switching and routing, quality of services, network security, content networking and embedded hardware software co-design. He can be reached via e-mail at ydlin@cs.nctu.edu.tw or through his Web site (www.cs.nctu.edu.tw/~ydlin).

Resources

John Lombardo, *Embedded Linux*, 1st ed., New Riders, July 5, 2001.

Todd Fischer, "Optimizing Embedded Linux", *Dr. Dobb's*, May 2002: **www.ddj.com/184405050**.

Lei Yang, Robert P. Dick, Haris Lekatsas and Srimat Chakradhar, "CRAMES: compressed RAM for embedded systems", International Conference on Hardware Software Codesign, Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, Jersey City, New Jersey, 2005, pp: 93–98.

"Buildroot—Usage and documentation v1.2", December 28, 2004: **buildroot.uclibc.org/buildroot.html**.

Karim Yaghmour, *Building Embedded Linux Systems*, 1st ed., O'Reilly, 2004.

AlphaMail Is Scalable and Accessible Web Mail

AlphaMail takes a unique approach to providing a Web-based IMAP client.

TONY KAY

AlphaMail is a high-performance, feature-rich, open-source Web mail system created at the University of Oregon. The interface includes message snippets in indexes, UTF8 composition and numerous viewers for attachments (such as image icon preview and file listings from tarballs). It also tries to strike a balance between desirable features and too much interface noise. It was created to address several problems that exist with other open-source and commercial Web mail systems.

Performance

The first concern AlphaMail addresses is performance. Almost all Web mail systems (such as Horde's IMP Web mail Client and SquirrelMail) use IMAP from within the Web server, which is incapable of persisting an IMAP session.

The IMAP protocol is designed to optimize access through persistent access, so this is an inherent and recognized problem. The problem is usually mitigated with an IMAP proxy that maintains a persistent connection. The problems with this solution are multifaceted.

One problem is that the code in the Web mail client itself cannot depend on the state of the IMAP connection and must repeat commands as if each mouse click were a new IMAP session. This is a problem, because the sequence of required events for a new session in the IMAP protocol include authenticating and selecting the desired folder. The benchmarks of several IMAP servers indicate that the repetition of the folder selection command, even if the folder is already authenticated and selected (that is, through a proxy), can cause significant extra server load.

These inefficiencies could be addressed through improvements in the IMAP server and proxy algorithms, but another problem is intractable: a proxy cannot improve the protocol. The fact that the Web mail client is using IMAP forces it to behave as a complete standalone client. If the developers want to add a complex feature, such as conversation views (à la Google mail), which requires complex message cross-referencing across several folders, the protocol itself becomes a major impediment.