CHALMERS

An analysis of how static and shared libraries affect memory usage on an IP-STB

Master of Science Thesis in the Programme Networks and Distributed Systems

Dongping Huang

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, September 2010

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

An analysis of how code locality affects the dynamic memory usage on an embedded set-top box running Linux

Dongping Huang

© Dongping Huang, September 2010.

Examiner: Björn von Sydow

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2010

Abstract

An IP-STB is a small and stand alone embedded system which is able to provide a series of advanced entertaining services. However, like an ordinary embedded system, an IP-STB still has to bear with extremely main memory constraints and no disk space. Normally, advanced features provided by an IP-STB require a large amount of limited main memory, therefore how to maintain a small static memory footprint and low memory usage becomes very important.

Libraries composed of shared code contribute to a significant amount of memory consumption. Libraries can be categorized into two types: static libraries and shared libraries. Both of them have their pros and cons in terms of memory usage and system performance. However, there is no existing generic method to determine when and why to choose one type over the other one.

To address this problem, a series of tools were implemented to give a quantitative analysis of how different types of libraries affect the system memory usage. Besides, their advantages and disadvantages were discussed and compared. In addition to the above, the optimization of shared libraries was also investigated.

Acknowledgement

First of all, I would like to express my deepest gratitude to Motorola for giving me this opportunity to accomplish this thesis. I would particularly like to acknowledge my supervisor Patrik Almqvist for his patient guidance and assistance.

Finally, I would like to thank my examiner Björn von Sydow at Chalmers University of Technology for his valuable guidance and feedback.

Table of Contents

1 Introduction

1.1 Background

An ordinary set-top box is a device that connects to a television to display digital TV and receives digital signal from external sources such as satellites, cables and antennas. An IP based set-top box (IP-STB), as its name implies, connects and receives digital signals from IP based networks. Taking full advantages of broadband networks, an IP-STB is able to provide a range of advanced services such as video on demand (VOD), time-shifting video, muliticast TV, as well as some Internet applications, including web surfing and so on.

The IP-STB used in this thesis is a small and stand alone embedded system based on the Renesas SH4 architecture with a limited amount of main memory (RAM) and flash memory, and lack of hard drive which means that no swap space is available. The software of the IP-STB (called KreaTV) is based on a Linux kernel with several processes running on top. Some of them are hardware specific but most of them are unaware of the underlying hardware and are Linux-generic and cross-compiled for all architectures.

1.2 Problems description

Despite the above hardware limitations, there is usually a requirement of heavy-weight user interfaces on top of web browsers or Java virtual machine which requires a large amount of limited memory, therefore it is important to maintain low memory usage: the more memory that can be saved, the more room there is for additional application functionality and capacity.

Shared code that makes up libraries contributes to a significant amount of the memory consumption, therefore, it is desirable to find a solution to optimize the use of shared code and minimize its memory footprint.

1.3 Objective

According to how shared code is shared among different applications, normally, libraries are categorized into two types: static libraries and shared libraries. Both of them have their pros and cons in terms of memory usage and system performance.

The main goal of this thesis work is to analyse shared code with respect to memory usage, and to propose a generic method that explains when and why to choose one type of libraries over the other one.

The secondary goal is to investigate the current GNU compiler and linker optimization options, and to propose a better compilation and linking strategy to generate faster and smaller code.

1.4 Glossary

ELF
: Executable and Linking Format, is the de-facto standard object file format in Linux.

DRAM
: Dynamic Random Access Memory

GNU
: *GNU* is a recursive acronym for *"GNU's Not Unix!"*. Programs released under the auspices of the GNU Project are called *GNU packages* or *GNU programs*.

GOT
: Global Offset Table, is an ELF data structure used to hold absolute virtual addresses of imported functions and variables.

IIP
: Installation Package, is a Motorola software package used to store software and configurations.

PIC
: Position Independent Code, is designed to load code in arbitrary location in the virtual memory without modification of the text segment of code. Typically, used for shared libraries.

PLT
: Procedure Linkage Table, is an ELF data structure used to redirect PIC function calls to GOT.

Pss
: Proportional Set Size, is a metric which reports the amount of physical memory that is used by a process. It is calculated by dividing each page by the number of processes sharing it.

Rss
: Resident Set Size, is a metric which reports the amount of virtual memory that is currently marked as resident in RAM.

RISC
: Reduced Instruction Set Computing, is a CPU design philosophy with few and simple instructions.

Squashfs
: Squash file system, is a read-only compressed file system for Linux. Typically, used in embedded systems.

1.5 Thesis organization

The remainder of this thesis is organized as below:

Chapter 2 provides a brief description of the hardware and software of the IP-STB as well as characteristics of tmpfs.

Chapter 3 gives a general description of libraries and tools.

Chapter 4 describes our methodology of how to measure the system memory usage and how to determine the impact on the run-time memory usage from static and shared libraires.

Chapter 5 gives a specific disscussion of advantages and disadvantages of static and shared libraries. A comparison of two types of libraries is provided. In addition, the topics of how to remove unused material and the benifits of position independent code are mentioned as well.

Chapter 6 concludes the thesis along with a discussion of future work.

2 The System

2.1 Hardware

The specific IP-STB used in this project is the 1920-9C version of Motorola VIP1900 series. The microprocessor used by this IP-STB is a RISC processor with integrated TV video processing and a MPEG decoder. It has 4 MB of NOR flash memory that is used for storing firmware and configuration information. 32 MB of NAND flash memory is used for storing the boot image. The size of the dynamic random-access-memory (DRAM) is 128 MB [1].

The interaction between users and the box is done through a remote control, in some configurations a keyboard is also supported.

2.2 Software

The KreaTV software is mainly written in C++ and based on open source software with Linux as the operating system kernel and runs on multiple hardware and architectures, e.g., the multiple IP-STB series. The architecture of software is designed for a hardware environment with very limited resources (memory and CPU). It is also designed for a specific user environment (the TV environment) and specific applications (broadcast television, video on demand, music, e-mail and chat) [2].

Due to the hardware limitations, the size of software becomes crucial. Several technologies and software specialized for embedded systems are employed to optimize software size. For instance, BusyBox [3] is utilized. BusyBox provides more than 70 core utilities within one application. Many of these utilities are reduced version with limited features, but still sufficient for most common needs. It can also be compiled to only contain necessary utilities that you select. Compression is another efficient technology to reduce the size of software, therefore it is widely used typically in the construction of boot images.

2.3 IP-STB installation packages

Applications and libraries as well as configuration files are organized and placed into different IP-STB installation packages (IIPs) according to different software modules. An IIP is a gzip compressed tar archive which could be installed in a boot image at build-time or run-time. An IIP may depend on another IIP, therefore, the dependencies between IIPs must be specified in order to make sure all IIPs are installed in the correct order. Different IIPs are combined to form a complete boot image [2].

2.4 Boot images

An boot image consists of a Linux kernel image and a root disk image (a tmpfs image) which is a gzip compressed cpio archive. A boot image itself is signed and encrypted, moreover, it is compressed as well. The usr_bin, usr_lib and usr_appliction folders in the /usr directory of the

root file system are compressed respectively into three squash file systems: usr_bin.sqfs, usr_lib.sqfs and usr_applications.sqfs. The boot image structure is shown as below:

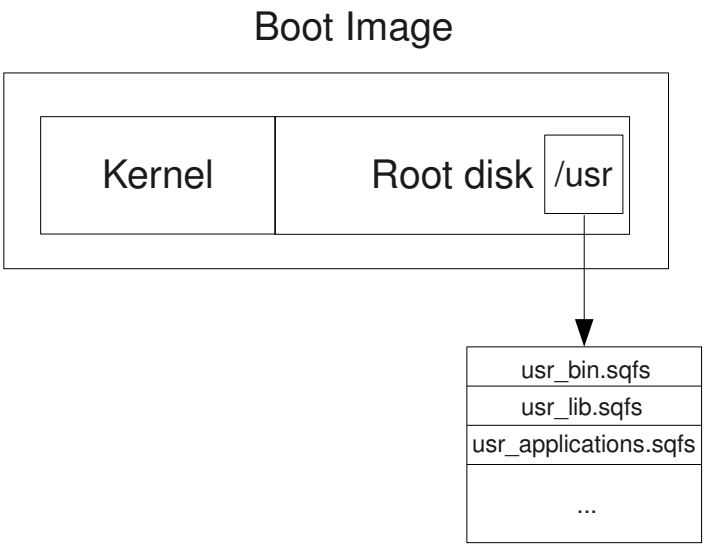

Figure 1. The boot image structure

2.5 Boot procedure

There are several methods to boot an IP-STB. In a live deployment, an Infocast Server is commonly used to transmit boot images to IP-STBs via multicast, however this is not the most convenient way when you are developing and testing IP-STB software in a lab environment, because it requires you to modify the Infocast Server settings. In this project, another easier method, TFTP booting is chosen. A TFTP server must be configured in the workstation in order to enable TFTP booting.

A boot image is downloaded into the IP-STB's flash memory from the workstation. Then the boot loader starts to execute, decrypts and decompresses the boot image, and loads the Linux kernel image and root disk image into RAM. Once Linux boots up, it decompresses and unpacks the root disk, and then mounts it as the root file system. The squash file systems are mounted as well.

2.6 File systems

In this system, tmpfs is used as the final root file system. Tmpfs is supported by Linux kernel from version 2.4 and up [4]. Tmpfs is a file system which keeps all files in virtual memory. Tmpfs puts everything into the kernel internal caches and grows and shrinks to accommodate the files it contains and is able to swap unneeded pages out to swap space. Since tmpfs lives completely in the page cache and on swap, all tmpfs pages currently in memory will show up as cached [5].

As we mentioned above, squash file system (Squashfs) is used as well. Squashfs is a compressed read-only file system for Linux [6]. Squash file system provides greater compression ratio and performance in comparison with other read-only compressed file systems. Besides, since the squash file system is read-only, using it can prevent malicious users from compromising system security effectively.

There are two facts which have close connections to memory usage and must be pointed out. Since tmpfs is a RAM-based root file system, the size change of applications and libraries would affect memory usage directly. In other words, the larger the file becomes, the more memory it will occupy. Another important fact is that when a process attempts to access a page which resides in the squash file systems, the appropriate page is decompressed and loaded into RAM. Because the squash file systems have already resided in RAM, this operation implies that there are two copies of each referenced page from the squash file systems in RAM: one compressed copy and one decompressed copy. We must always keep these two facts in mind. Figure 2 shows the process of populating and mounting file systems.

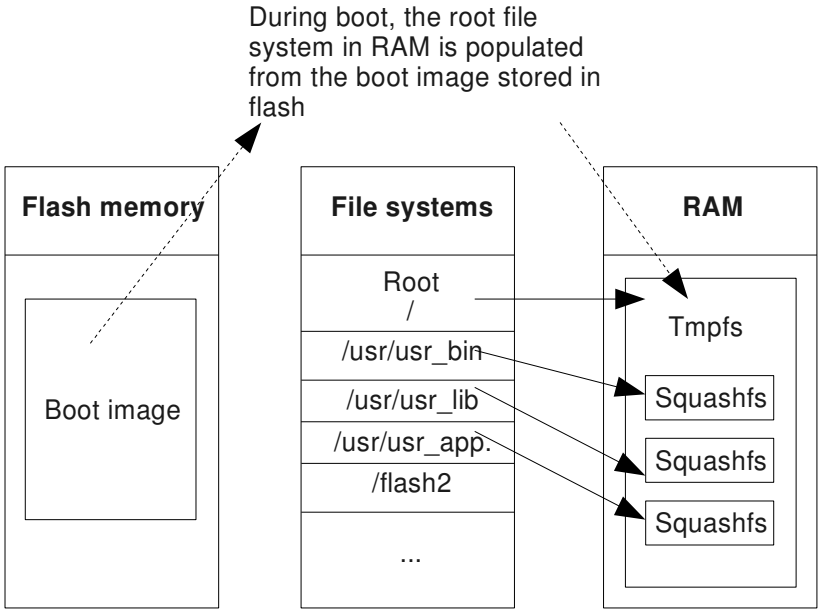

Figure 2. The process of populating and mounting file systems

3 Libraries and tools

3.1 ELF

The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI) [7]. Currently, ELF is the de-facto standard object file format in Linux. According to different usages, ELF object files can be categorized into the following three main types:

1. A relocatable file holds code and data suitable for linking with other object files to create an executable or a shared object file. Compiling a source file generates a relocatable file. This file has the file extension ".o".

2. An executable file holds a program suitable for execution. Normally, this kind of file does not have a file extension.

3. A shared object file holds code and data suitable for linking in two contexts. First, the link editor may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image. The file extension of a shared object file is ".so" [7].

Object files are binary representations of programs which are intended to run directly. Object files participate in both program linking and execution processes. Corresponding to these two activities, there are two parallel views of object files' contents: linking view and execution view. The following figure shows these two views.

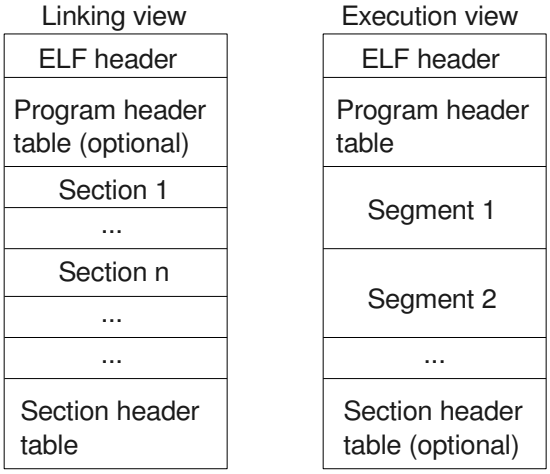

Figure 3. Linking view and execution view [7]

From figure 3 we can see, the ELF header resides in the beginning of an object file. It holds information about file's organization and identification. The program header table, if present, tells the system how to create a process memory image. The section header contains information about various sections, such as lengths of sections, section names, access permissions and so on. A section is the smallest indivisible unit that can be processed within an ELF file. Each section contains a single type of information, such as program instructions, data, symbols, relocation information and so on. A segment is a collection of sections, it is the smallest indivisible unit that can be mapped into the memory image of a process. Loadable segments such as the text segment and data segment must be mapped. There are dozens of sections within an ELF file, some of them contain debug information can be stripped away for production, here, we only select and explain the following important sections:

.text

This section holds actual executable instructions.

.bss

This section holds uninitialized data such as uninitialized global variables and local static variables. It occupies no file space. When a program starts to run, this section will be initialized with zero.

.data

This section holds initialized data such as initialized global variables and local static variables. If the initial value of data is equal to zero, this data will be placed in the .bss section.

.rodata

This section holds read-only data such as string constants.

.dynsym

This sections holds the dynamic linking symbol table which contains symbols participating in dynamic linking.

.dynstr

This section holds strings that are needed in dynamic linking. These strings represent the names associated with entries of dynamic linking symbol table.

.plt

This section holds the procedure linkage table. The PLT entries convert position-independent function calls to absolute virtual addresses.

.got

This section holds the global offset table. The GOT entries contain absolute virtual addresses of each imported variables and functions.

.gnu.hash

This section holds the GNU style hash table which is used to facilitate symbol look up.

.rela.dyn

This section holds relocation records correspond to imported variables.

.rela.plt

This section contains relocation records correspond to imported functions.

3.2 Static libraries

To improve modularity and reusability, normally, commonly used functions are included in libraries. The traditional library is the static library which is a collection of concatenated object files of similar types. A static library is stored on disk as an archive (.a file). There is an index which provides an association of symbols with the object files that supply symbol definitions in a static library. This index will be used by the link-editor (ld) to speed up symbol look up. The ranlib command can be used to construct this index, it is equivalent to passing the -s option to the ar command.

When a static library is involved in linking, by default, the link-editor performs selective extraction [7] of object files in the static library. It passes through the library iteratively, and only extracts the object files which contain symbol definitions of undefined symbols in the link-editor's internal symbol table. The link-editor concatenates all program data sections such as .text, .data, and .bss sections of selected object files to form new sections in the later output object file. The link information sections such as .symtab, .synstr sections and so on will be used by the link-editor to modify other sections. These information sections are also used to generate new link information sections in the later output object file. Figure 4 shows the process of linking object files against a static library as well as generating an executable file and a shared library.

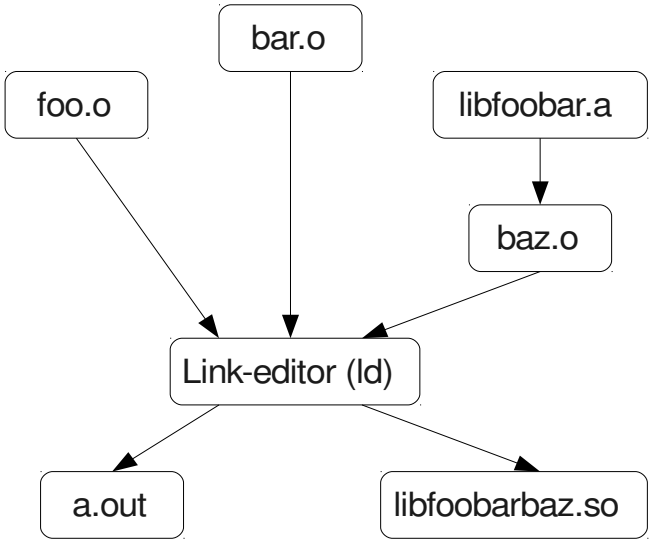

Figure 4. Linking process of static libraries

3.3 Shared libraries

In order to address resource waste and maintenance problems with static libraries, shared libraries are utilized by most operating system. The idea behind shared libraries is to have only one copy of commonly used functions in physical memory, different applications can share these functions instead of having their own copies. Shared libraries are often referred to as shared objects.

A shared library is an indivisible unit that is created from one or more relocatable object files. When a shared library is involved in linking, unlike linking against a static library, the shared library's program data sections and most of the link information sections will not be used by the link-editor. However, some other information will be generated to record the shared library as a dependency of the output object file. This notable difference between linking against static and shared libraries indicates that the size of executable files can be reduced significantly by linking against shared libraries, thus disk space is saved. Another primary difference is that the actual linking process of a shared library is carried out at run-time by the run-time linker (ld-linux.so) which itself is a shared library. The run-time linker is responsible for loading shared libraries and binding symbols to applications before the applications gain control from operating systems to execute. The run-time linker can also be called during the process of an application's execution to load additional shared libraries by using dlopen function. Symbols provided by these shared libraries can be bound by using symbols dlsym function, and this process is often referred to as dynamic loading. Once a shared library is loaded in physical memory, it's functions and variables can be shared by all processes that reference it, therefore the amount of used physical memory can be reduced.

To build an efficient shared library, a special option (-fPIC/-fpic) should be passed to the compiler. The benefits of using this option will be mentioned in the a section in detail. Figure 5 shows the process of linking object files against a shared library as well as generating an executable files and a new shared library.

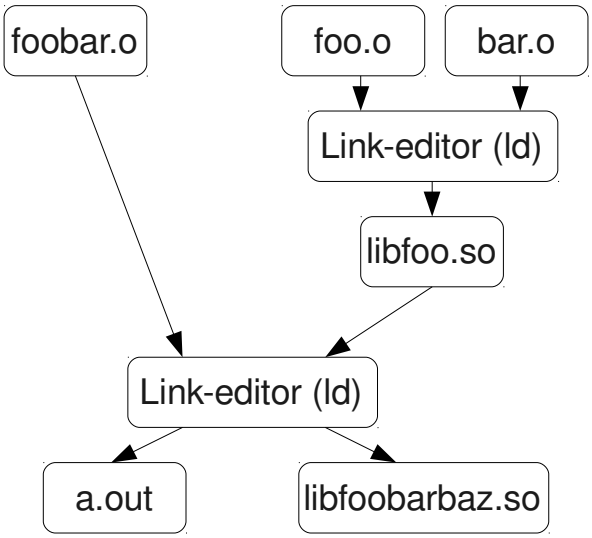

Figure 5. Linking process of shared libraries

3.4 Linux memory mapping

During the initial start-up of a program, the contents of its ELF executable file must be mapped into the corresponding process' virtual address space by the Linux kernel. This is also true for any shared library that the executable file has been linked to. However, the memory mapping of shared libraries is performed by the run-time linker instead of the kernel. The address space of a process is represented by a mm_struct data structure. The contents of the address space are defined by a list of vm_area_struct structures, each structure represents a virtual memory region corresponding to a loadable segment of an ELF object file: the text and data segment. For reason of efficiency, both the initial address and the length of a memory region must be a multiple of 4096 bytes, which is the default Linux page size.

3.5 GNU compiler and binutils

GCC is the GNU Compiler Collection, which in our version currently contains compilers for C and C++. GNU binutils is a collection of binary utilities used for manipulation of object files. The compiler and binary utilities included in the KreaTV toolchain is a cross platform toolchain. It is built to run on a normal Linux PC and is able to generate binaries for various IP-STB series. Here we only list the following binary utilities which are useful for our project.

ld
: The GNU link editor

as
: The GNU assembler

ar
: This utility is used for creating, modifying and extracting from archives. It can be used for generation of static libraries.

c++filt
: This utility is a de-mangling filter for C++ symbols.

ranlib
: This utility is used for creating indexes for archives.

readelf
: This utility is heavily used in our project. It is used to display contents of ELF object files. Combining with various options, a lot of useful information such as the ELF header, sections, symbols and so on can be read.

size
: This utility tells the text segment size, data segment size, .bss section size and total size of an object file. Note, the data segment in this context is subtly different from the notion we mentioned previously. In this context, the size of .bss section is listed separately, therefore, the size of the data segment is equal to the size of former data segment minus the size of .bss section.

3.6 The /proc file system

The /proc file system is a virtual file system that resides in RAM. It does not contain any "physical" files (most of them are zero byte in size) but very valuable run-time system information such as the state of the kernel, the state of individual process and so on. In this project, two of them are utilized. The first file is /proc/meminfo that reports plenty of information about the current RAM usage on the system. Figure 6 is a snapshot of the output from a sample /proc/meminfo file.

MemTotal:	71656	kB
MemFree:	10032	kB
Buffers:	8068	kB
Cached:	35680	kB
SwapCached:	0	kB
Active:	12880	kB
Inactive:	37040	kB
SwapTotal:	0	kB
SwapFree:	0	kB
Dirty:	0	kB
Writeback:	0	kB
AnonPages:	6188	kB
Mapped:	13004	kB
Slab:	5140	kB
Sreclaimable:	1132	kB
Sunreclaim:	4008	kB
PageTables:	456	kB
NFS_Unstable:	0	kB
Bounce:	0	kB
CommitLimit:	35828	kB
Committed_AS:	116152	kB
VmallocTotal:	507880	kB
VmallocUsed:	13352	kB
VmallocChunk:	493160	kB

Figure 6. Information reported from a /proc/meminfo file

From figure 6 we can see that, a lot of memory details are revealed. Here, we only interpret the most important metrics.

MemTotal

Amount of usable RAM (i.e. total amount of RAM minus a few reserved bits and the kernel binary code).

MemFree

Amount of RAM unused by the system, in kilobytes.

Buffers

Relatively temporary storage in RAM for raw disk blocks. After the Linux kernel version 2.4.10, buffers do not exist any more. In fact, they have been merged into page cache [8].

Cached

Cache in RAM for pages read from disk. Buffers and swap cache are excluded.

As we have mentioned previously, there is no swap space available in our system, therefore swap associated metrics such as SwapCached, SwapTotal and SwapFree are equal to zero.

Another file is /proc/PID/smaps that displays information of memory regions of a program and its dependent shared libraries from the process' virtual address space as well as specific physical memory consumption information. The PID indicates the specific process id. For each process, a series of information is shown as the following:

```
00400000-00427000 r-xp 00000000 07:00 67      /usr/bin_ro/procman
Size:                156  kB
Rss:                 140  kB
Pss:                 140  kB
Shared_Clean:        0    kB
Shared_Dirty:        0    kB
Private_Clean:       140  kB
Private_Dirty:       0    kB
Referenced:          140  kB
00437000-00438000 rw-p 00027000 07:00 67      /usr/bin_ro/procman
Size:                4    kB
Rss:                 4    kB
Pss:                 4    kB
Shared_Clean:        0    kB
Shared_Dirty:        0    kB
Private_Clean:       0    kB
Private_Dirty:       4    kB
Referenced:          4    kB
...
29580000-29589000 r-xp 00000000 07:01 27      /usr/lib_ro/libemio.so
Size:                36   kB
Rss:                 32   kB
Pss:                 10   kB
Shared_Clean:        32   kB
```

```
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         0 kB
Referenced:           32 kB
...
```

Here, we omit some parts of the output, the rest are similar to what is shown. As we can see, for each mapping, there are a few lines of information. The first line shows the virtual memory address, access permission, offset into the virtual memory address, device number, inode number and the path name of the associated file for this mapping. The Size indicates the amount of virtual memory for this mapping, commonly another term, the Vss (virtual set size), is used. The Rss (resident set size) reports the amount of virtual memory that is currently marked as resident in RAM, in other words, the amount of used physical memory for this mapping. The Rss can be misleading for shared pages, because it counts the amount of shared pages as if there was only a single process using them. For example, if three processes share 30 pages of a shared library, this shared library will contribute 30 pages to the Rss that is reported for each of these three processes. The Pss (proportional set size) addresses this problem well. It is calculated by dividing each shared page by the number of processes sharing it, therefore, for the previous example, the shared library will only contribute 10 pages to the Pss instead. Moreover, when one of these processes is terminated, these 30 pages will be proportionally distributed to the Pss of the remaining two processes that are still using the shared library. The Pss is an extremely useful metric, because it presents an accurate method to measure the memory usage of each process. If we sum up all the Pss values from a running process' smaps file, we are able to find out how much physical memory this process is using currently. The Pss was introduced since the Linux kernel version of 2.6.25 [9]. However, the kernel of the IP-STB did not support this metric initially. Fortunately, applying a Pss patch to the kernel is quite easy. The remaining lines show the number of clean and dirty shared pages in the mapping, and the number of clean and dirty private pages in the mapping. The Referenced indicates the amount of memory currently marked as referenced or accessed.

The mapping with r-xp access permission (i.e., read, execute and private permission) corresponds to the text segment of the executable file or shared library. Likewise, the mapping with rw-p access permission (i.e., read, write and private permission) corresponds to the data segment of the executable file or shared library.

4 Measuring memory usage: methodology

In order to analyse how static and shared libraries affect physical memory usage and which type of libraries is preferred, the most efficient and straightforward method is to convert a given library, either change a static library to a shared library, or convert a shared library to a static one, then link all associated applications and shared libraries which have dependencies on the original given library against the newly created library. Next, boot up the system and record memory consumption. Last, observe and compare memory usage variations, then determine which version is more memory efficient for the given library.

4.1 Prerequisites

Before we set our hands to measure physical memory usage, there are three essential prerequisites:

First, we must find a way to list the dependency relationship among applications, static libraries and shared libraries. In practice, almost every application has dependencies on various static libraries and shared libraries. A shared library sometimes also has its own dependencies as well, it may need functions and variables that are defined in other shared libraries or static libraries.

Second, accurate memory usage statistics must be obtained to serve as the benchmarks for comparison. In the absence of accurate benchmarks, it is impossible to correctly assess the effect on physical memory usage by converting libraries.

Third, how to measure physical memory usage accurately is the most important issue. It is closely connected with the second prerequisite.

We will describe how to address these prerequisites in the following subsections respectively.

4.2 Listing dependencies

When we compiled the whole source file tree, by passing the -t option to the link-editor, the link-editor would print the name of files it processed when it was linking an application or a shared library. Therefore, we could observe which shared libraries and static libraries were used by each application or shared library. The whole compilation process took several hours to complete and the tremendous output from the link-editor was saved in a log file. The log file contained a hundred thousand lines of information (both linking and other building information), obviously, it was quite time consuming and error prone to analyse log file manually. Consequently, a Bash script was written to facilitate the analysis and generate a dependency report automatically. The resulting dependencies were listed and sorted in the form as below:

[library name]
        [list of files that use the library]

lib1.a

lib2.so
lib3.so
exe1
...

lib1.so

lib2.so
lib4.so
exe2
...

The libraries were divided into two categories (shared and static libraries). For each category, libraries were sequenced in descending order of the number of files that used them. As a result, it was very convenient to find out which shared or static library was used by the most number of files.

4.3 Measuring memory usage

As we noted above, the second and the third prerequisite are closely connected. Once the issue of how to measure memory usage is solved, then accurate memory usage statistics can be obtained. First of all, the term of a stable system state must be defined, because any change of file size or running additional service will affect the system memory usage statistics directly. In order to eliminate the effect of running additional service, the definition of the stable system is: after the system boots up, the system becomes stable with only the default processes running on it. All the benchmarks must be measured from the initial stable system.

Initially, the measurement of memory usage was performed from the whole system's point of view: the value of the metric MemFree of the /proc/meminfo file from the original system served as the benchmark for comparison. However, in the initial stage of implementation, this benchmark was proved to be problematic and even misleading. The following chart shows the unstable values of MemFree measured eight times in the same time interval.

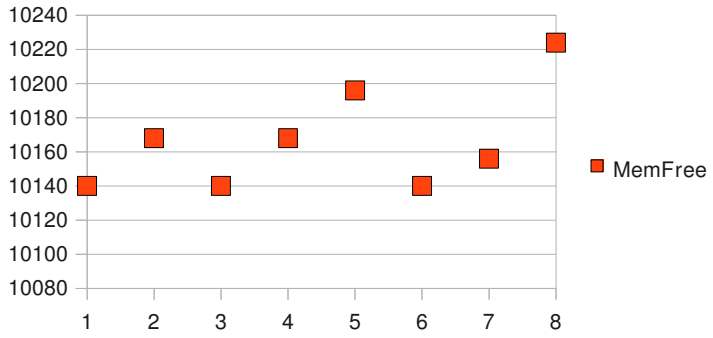

Figure 7. Variations of the MemFree values

Due to its inaccuracy, the MemFree was not the desired candidate for the benchmark. Another approach was tested, there was a running background daemon which monitored the amount of available physical memory, once this amount reached a pre-set threshold, it would trigger the system to reboot. The program allocmem which allocated a certain amount of physical memory every millisecond was used to eat up the available physical memory. In theory, as soon as the system crashed, the amount of the allocated physical memory should be equal to the total amount of available physical memory minus the threshold. Therefore, as long as the amount of the allocated physical memory was constant, it was qualified for becoming to the benchmark. Unfortunately, after a lot of tests, this amount was proved to be unstable and unreliable as well. Consequently, this approach was discarded. Since there were not any feasible methods to measure physical memory from the whole system's perspective, we were forced to switch our train of thought to perform the measurement from the process' point of view. As we mentioned in the previous chapter, the sum of all the Pss values from a running process' smaps file (/proc/pid/smaps) is an accurate representation of the physical memory usage for a single process, more importantly, the statistics from the /proc/pid/smaps file were verified to be stable. Besides, the number of default processes is fixed. Thus, the total amount of physical memory used by the default running processes can be obtained by summing up all the sum of Pss of each process, i.e., $\sum_{k=1}^{n}(\sum_{i=1}^{n} Pss_i)_k$ Here, k indicates the number of default processes, i indicates the number of Pss values from a single process. It is noted that the original Pss is calculated by dividing each shared page by the number of processes sharing it, in order to keep division error low, the PAGE_SIZE (1000000000000 B) is left shifted 12 bits of error bits before the division, however, the resulting Pss is right shifted of 22 bits (12+10) to align with 1 kB which is the minimum output unit of /proc/pid/smaps. 22 bits of precision are thrown away merely for alignment. In order to keep division error as low as possible, the original Pss was modified. Instead of 22 bits, 12 bits was thrown away, this gave us the real byte count for Pss. The specific code of the modified Pss patch is shown in the appendix A.

On the other hand, the conversion of libraries will affect not only the process' memory usage but also the size of associated executable files and shared libraries; accordingly, the size of the root disk which is loaded in RAM will be affected as well. To be more specific, in the case of changing a static library to a shared library, the size of associated files will increase. Besides this shared version of the library must be available at run-time which implies it must be copied to the root disk; in the case of converting a shared library to a static library, the size of associated files will decrease and the original shared library must be removed from the root disk, since it becomes useless. In addition, all the user space files are compressed and placed into the squash file systems in the root disk. Consequently, the impact on memory usage from file size changes will be reflected on the difference between the size of initial root disk and new one. The most precise and convenient method to get the size of root disk is finding the size of tmpfs by means of the df command on the system. As we noted before, the root disk is mounted on the system as a tmpfs, therefore, the size of tmpfs must be equal to the size of the root disk. The df command used to show the file system disk space usage in kilobytes, in our case, it shows the RAM usage of file systems.

Based on the above analysis, the memory usage can be measured through the following formula from the system:

$$tmpfs + (\sum_{k=1}^{n} (\sum_{i=1}^{n} Pss_i)_k)$$

Here, the *tmpfs* denotes the size of tmpfs measured from the initial stable system. Accordingly, the benchmark can be obtained by means of the above formula on the initial stable system, i.e., $tmpfs_{init} + (\sum_{k=1}^{n} (\sum_{i=1}^{n} Pss_i)_k)_{init}$

With the benchmark on hand, another formula was proposed to determine which version of a given library was more memory efficient. It is shown as below:

$$tmpfs_{init} + (\sum_{k=1}^{n} (\sum_{i=1}^{n} Pss_i)_k)_{init} - (tmpfs_{new} + (\sum_{k=1}^{n} (\sum_{i=1}^{n} Pss_i)_k)_{new})$$

Each time, the subtrahend is re-measured after every conversion of a given library. If the difference is positive, this means the new version of the library is more memory efficient than the original one, otherwise, the original one is preferred. Two scripts were written to facilitate analysis. The fist script is mem_usage_gen.sh used to record the /proc/pid/smaps files and generate a memory usage report on the system. Another script is mem_cmp_gen.sh used to compare the new memory usage report with the benchmarks, then generate a comparison report in the form as below:

Process		Vss(kB)	Rss(kB)	Pss(B)
process1	xxx	xxx	xxx	
		xxx	xxx	xxx
...				
process(n)	xxx	xxx	xxx	
		xxx	xxx	xxx

we lost/gain xx kB memory.

4.4 The specific procedure of measurement

In the case of a static library, the specific procedure of measurement can be summarized into the following steps:

1. Choose a static library, then recompile the source codes composing the library, last, re-link them to a shared library.

2. Re-link all applications and shared libraries which depend on this static library.

3. Copy the new shared library to an IIP, since it must be available at run-time, and then rebuild the rest of associated IIPs which contains the changed applications and shared libraries, next construct a new boot image and load it into the IP-STB.

4. Set up a telnet connection to the IP-STB, run the mem_usage_gen.sh script with the library name as an argument to generate a memory usage report with the suffix of the library name.

5. Copy the memory usage report from the IP-STB to the workstation, then run the mem_cmp_gen.sh , pass the file name of benchmark as the first argument and the file name of the new memory usage report as the second argument to generate a comparison report.

In contrary to a static library, there is a small difference in step 3 when we convert a shared library to a static one. Instead of copying a new library to an IIP, the original shared library contained in the IIP must be removed. The specific results of converting libraries are shown in appendix B.

5 Discussion

5.1 Shortcomings of static libraries

There is a common misunderstanding about how the link-editor concatenates program data sections. People always take it for granted that the link-editor only copies referenced symbols into the output object file. However, as we mentioned above, concatenation of all program data sections indicates that if the selected object file contains both referenced and non-referenced symbols (unused symbols), unfortunately, the code and data associated with non-referenced symbols will also be copied into the later output object file. This unused data not only bloats the output object file but also puts an extra burden on the link-editor. Several experiments were carried out in order to verify the above viewpoint. Each sample library was built into two versions: one static version and one shared version. By linking the sample executable file against static and shared versions of a sample library respectively, two versions of a sample executable file were obtained. Figure 8 shows the symbol table entries which associated with the static version of a specific sample library from the static version of a specific sample executable file. Figure 9 shows the symbol table entries which associated with the shared version of the same sample library from the static version of the sample executable file. Obviously, the static version of the sample executable file contains much more symbols than the shared version. To be more precisely, the static version contains all symbols that are defined in the TEdidParser.cpp source file.

```
[xhbf63@annika-anka audiooutput]$ readelf -Ws st40/audiooutput | grep -A 100000 -E '.symtab' | grep TEdidParser
   103: 00000000     0 FILE    LOCAL  DEFAULT  ABS TEdidParser.cpp
   104: 00413040    24 FUNC    LOCAL  DEFAULT   11 _GLOBAL__I__ZN11TEdidParser15EDID_BLOCK_SIZEE
   105: 0041600c    70 OBJECT  LOCAL  DEFAULT   13 _ZZN11TEdidParser17GetNumberOfBlocksEPKhRhE19__PRETTY_FUNCTION__
   106: 00416054    34 OBJECT  LOCAL  DEFAULT   13 _ZZN11TEdidParser5IsDviEvE19__PRETTY_FUNCTION__
   108: 00415f38   157 OBJECT  LOCAL  DEFAULT   13 _ZZN11TEdidParser24GetShortVideoDescriptorsEvE19__PRETTY_FUNCTION__
   109: 00415e98   157 OBJECT  LOCAL  DEFAULT   13 _ZZN11TEdidParser24GetShortAudioDescriptorsEvE19__PRETTY_FUNCTION__
   110: 00415e10   133 OBJECT  LOCAL  DEFAULT   13 _ZZN11TEdidParser14GetColorSpacesEvE19__PRETTY_FUNCTION__
   111: 00415fd8    51 OBJECT  LOCAL  DEFAULT   13 _ZZN11TEdidParser8LoadEdidEPKhE19__PRETTY_FUNCTION__
   127: 0042c358    12 OBJECT  WEAK   DEFAULT   20 _ZTI11TEdidParser
   138: 004132c4   176 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser24FillShortAudioDescriptorEPKhj
   142: 00412d3c    30 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser15OneByteCheckSumEPKhj
   143: 00413058   116 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser24GetShortVideoDescriptorsEv
   180: 004131b4   272 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser24FillShortVideoDescriptorEPKhj
   202: 00413768   140 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParserD1Ev
   219: 00413660   132 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParserC1Ev
   290: 00412e44    96 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser10ResetStateEv
   321: 0042c8c0     4 OBJECT  GLOBAL DEFAULT   25 _ZN11TEdidParser22VENDOR_ID_HDMI_UNKNOWNE
   328: 0042c328    48 OBJECT  WEAK   DEFAULT   20 _ZTV11TEdidParser
   386: 00412e00    16 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser11IsValidEdidEv
   435: 004137f4   140 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParserD2Ev
   469: 00412ea4   112 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser17GetNumberOfBlocksEPKhRh
   489: 00415e0c     4 OBJECT  GLOBAL DEFAULT   13 _ZN11TEdidParser15EDID_BLOCK_SIZEE
   525: 00413140   116 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser14GetColorSpacesEv
   533: 0042c8c4     4 OBJECT  GLOBAL DEFAULT   25 _ZN11TEdidParser23VENDOR_ID_SCART_UNKNOWNE
   658: 00412f14    88 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser5IsDviEv
   669: 004130cc   116 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser24GetShortAudioDescriptorsEv
   671: 00413374   598 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser8LoadEdidEPKh
   706: 004136e4   132 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParserC2Ev
   725: 00412e10    52 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser11GetVendorIdEv
   752: 004135ca   150 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParserD0Ev
   762: 00412d5a   166 FUNC    GLOBAL DEFAULT   11 _ZN11TEdidParser18ValidateFirstBlockEPKh
```

Figure 8. Symbol table entries from the static version of the executable file

```
[xhbf63@annika-anka audiooutput]$ readelf -Ws st40/audiooutput | grep -A 100000 -E '.symtab' | grep TEdidParser
   183: 004063b8   140 FUNC    GLOBAL DEFAULT  UND _ZN11TEdidParserD1Ev
   198: 00406428   132 FUNC    GLOBAL DEFAULT  UND _ZN11TEdidParserC1Ev
   426: 00406888   112 FUNC    GLOBAL DEFAULT  UND _ZN11TEdidParser17GetNumberOfBlocksEPKhRh
   603: 00406bd0    88 FUNC    GLOBAL DEFAULT  UND _ZN11TEdidParser5IsDviEv
   613: 00406c78   116 FUNC    GLOBAL DEFAULT  UND _ZN11TEdidParser24GetShortAudioDescriptorsEv
   615: 00406cb0   598 FUNC    GLOBAL DEFAULT  UND _ZN11TEdidParser8LoadEdidEPKh
```

Figure 9. Symbol table entries from the shared version of the executable file

On the other hand, in case of each executable file linked against a static library, it must have its individual copy of shared code, the duplicated code will lead to a significant resource waste in terms of disk space and physical memory. Imagine in a common Linux system, if each object file has its own copy of printf function of the C library, the required amount of disk space will increase dramatically. Moreover, it may take up more physical memory typically where there are a number of concurrently running processes using common functions of a static library. Another disadvantage of static libraries is that whenever a static library is modified or updated, everything linked against this static library must be recompiled in order to make the changes take effect. This will bring a lot of troubles to the product update, installation and release. For instance, one program consists of 20 modules, each of them is 1 MB, whenever any module is updated, a user has to retrieve this 20 MB program.

5.2 Shortcomings of shared libraries

Despite the disadvantages of static libraries, shared libraries are not a panacea to shared code issues. There are still several shortcomings of them. First, there is usually a trade-off in performance, as we mentioned above, shared libraries are loaded at run-time by the run-time liker, resolving and relocating symbols at run-time will lead a considerable one-time performance degradation when a program starts up, since the program can not gain control to execute until the run-time linker completes its tasks. Second, although the text segment of a shared library can be shared by multiple processes, the data segment is private to each individual process. Thus each process has to maintain its own copy of the data segment of a shared library in RAM. Ordinarily the data segment is relative small. However, since the default page size is 4 kB, which means the minimum physical memory requirement is 4 kB due to the page alignment regardless how small the data segment it is. Third, there is a potential memory run-time overhead of using a shared library. During our implementation phase, from the /proc/pid/smaps files we observed that: when converted a static library to a shared one, the decrease of the Rss values of the text segments of associated processes were quite smaller than the Rss values of text segments of the newly introduced shared library, or in reverse conversion, the increase of the Rss values of the text segments were still quite smaller than the Rss values of the original shared library. It implies that a considerable amount of other information rather than the actual instructions (contained in the text section) of a shared library must also be loaded in RAM while the run-time linker loads it. Several experiments were carried out to make this run-time overhead more visible. The most straightforward method is to link an unused shared library against an executable file (the link-editor will record this shared library as a dependency of the executable file), and then execute

this executable file and check the Rss value of the text segment of the unused shared library from the /proc/pid/smaps of the process, since neither functions nor symbols of the chosen library will be used by the running process, this Rss value should be sufficient to represent the amount of the run-time overhead. The following chart shows the Rss values and the size of the sample shared libraries.

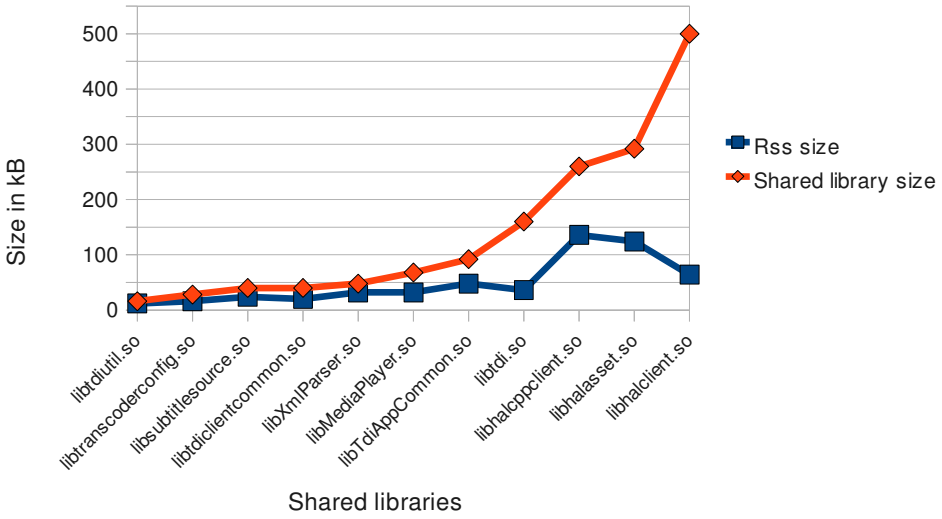

Figure 10. The size of run-time overhead and shared libraries

From the above figure we can find that the size of run-time overhead is considerably large in most cases. Besides, it is not in direct proportion to the size of shared libraries. Particularly, in comparison to the share library libhalcppclient.so which shows the 136 kB run-time overhead in 260 kB text segment, the size of the text segment of the shared library libhalclient.so is 500 kB, while its run-time overhead is only 64 kB. It is extraordinarily difficult to figure out the exact pages which compose the run-time overhead by means of the existing Binutils tools. However, according to the missions of the run-time linker during the application's startup phase, it is able to derive that several sections should be partially or entirely loaded in physical memory. Because, once the run-time linker has loaded all the dependencies required by an application, it processes each object file and performs all necessary relocation, symbol lookup and symbol resolution as well as application and dependencies initialization. The whole loading, relocation and initialization process proceeds in the following steps:

1. All the loaded objects make up a symbol lookup scope for the run-time linker. The run-time linker maintains a chain of symbol tables with pointers to its own dynamic symbol table and application's dynamic symbol table as well as all loaded shared libraries' dynamic symbol tables.

2. For each imported variable symbol, the run-time linker checks the associated relocation record in the executable file's .rela.dyn section (relocation records in the .rela.dyn section must be processed immediately after the loading phase), and determines the hash value for the relocation symbol. Next, in the first shared library of the look up scope, it uses the hash value to determine the hash bucket contained in the .gnu.hash section to find the symbol name contained in the .dynstr section. Then it compares the symbol name with the relocation name. If the names match, it

subsequently compares the versions of each symbol. If the versions match as well, then the definition of the relocation symbol is found. If the definition can not be found in the current shared library, the run-time linker proceeds with the next shared library in the lookup scope.

3. Once the definition of the relocation symbol is found, the run-time linker adds the absolute virtual address to the corresponding GOT entry.

The above relocation process is repeated for each relocation symbol in each loaded objects' .rela.dyn sections. Last, before transferring the control to the application, the run-time linker still needs to execute any .init sections found in the loaded dependencies.

According to the above description, we can conclude that sections such as the .gnu.hash, the .dynsym", the .dynstr, the .rela.dyn, the .gnu.version, the .gnu.version_r and the .init must be available at run-time and loaded in physical memory partially or entirely by the run-time linker. This implies that the run-time overhead may be composed of the above 6 sections. From figure 11 we can see that the variation trends of the two lines are quite similar. However, the sum of the size of the above 6 sections of each shared library is smaller than the size of run-time overhead. This may be due to the following reasons:

1. Without consideration of page alignment restraint, while calculating the sum of sections. It may lead to 4 kB to 8 kB skew.

2. The first few relocation records with the R_SH_RELATIVE type in the .rela.dyn section are associated with the contents of the .ctors data section which starts immediately after the .gcc_except_table text section. Therefore, the beginning part of the .ctors section may coexist with the end part of the .gcc_except_table section in the same page. When this page is referenced, it will be counted in both text and data segment, which will result in 4 kB skew.

3. Unknown reasons. Even counting the above two skews in, in some cases, the sums of the size of sections are still smaller than the measured run-time overhead.

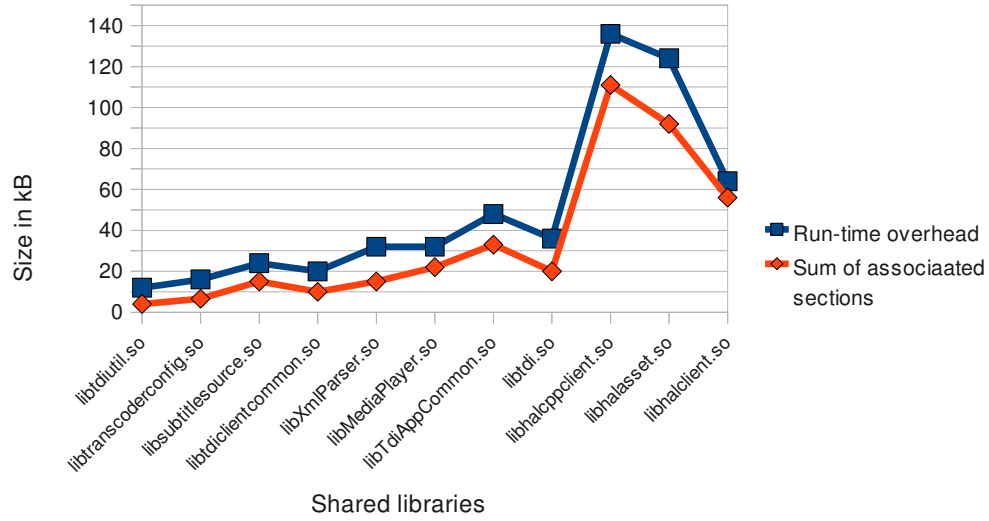

Figure 11. The variation trends of run-time overhead and sum of relocation associated sections

Figure 12 shows the comparison between the sum of associated sections and the measured run-time overhead, after including the first and second skews.

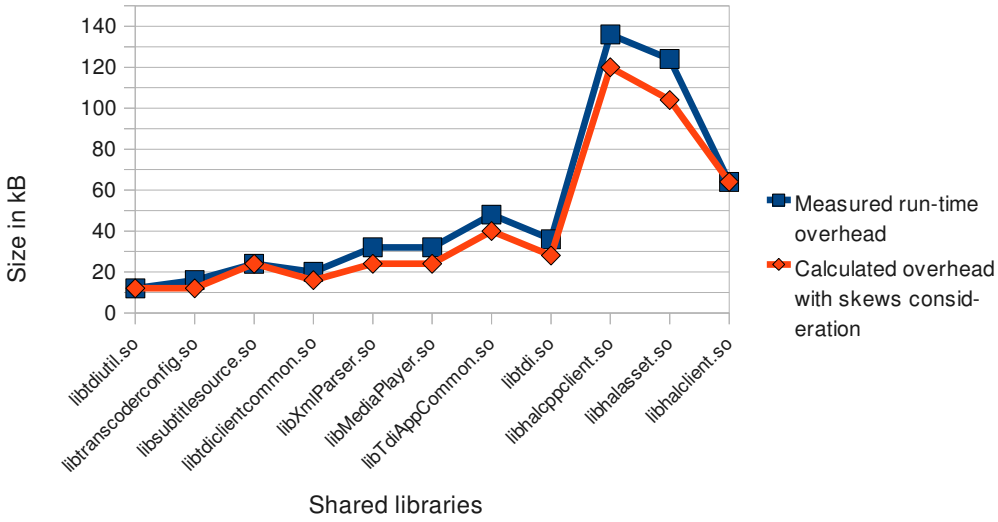

Figure 12. Comparison between the sum of associated sections and the measured run-time overhead, after including the first and second skews.

From the above figure we can see, in most cases, the differences between the measured run-time overhead and the calculated overhead are very small. Therefore, we believe that the calculated overhead with skews consideration should be sufficient to represent the size of run-time overhead approximately. Besides, it is noted that the .dynsym and .dynstr sections are the dominative factors to the run-time overhead, since they are very large in size, while the other 4 sections are relative small.

5.3 Static libraries or shared libraries

In the above two sections, we have discussed the shortcomings of static libraries and shared libraries respectively. In this section, we will give a general comparison of static libraries and shared libraries in terms of memory usage and system performance as well as compatibility and extendibility.

5.3.1 Memory usage

From the size of the root disk point of view, all code of referenced object files of a static library are merged into the resulting executable files, thus the file size of executable files are increased which in turn may lead to a lager root disk which requires more physical memory space. However, since shared library must be available as a whole unit at run-time which means it must be contained in the root disk, in some cases, the size of a shared library may outweigh the increased size of executable files due to statical linking. This depends on several factors such as how the object files of a library are shared by executable files, and the number of executable files sharing the library.

From run-time memory usage point of view, in theory, there is only one copy of commonly used functions or variables in physical memory, thereby if several applications use a common function or variable, physical memory will be saved without duplicated copies. However, as we stated in the section 5.2, using a shared library will introduce a considerable large amount of run-time overhead. Although this run-time overhead is shareable, in some cases, the run-time overhead may outweigh the memory saving from keeping one copy of function or variable in physical memory. We should also remember that using a shared library, each application should maintain at least 4 kB for the shared library's data segment which is not shareable. Since the data segment is relative small, there are always gaps existing in the physical memory pages corresponding to data segments, therefore, in some cases, information of the shared library's data segment can be merged into to the executable files' data segment thus fills up the gaps and keeps them still within the page boundary. Consequently, 4 kB physical memory could be saved for each application, if we use static library instead. A much larger effect is from additional dependencies. If an application references a shared library, all the dependencies of the shared library should be mapped and loaded, however, if a static version of the library is used, only the dependencies of the referenced object files must be loaded instead of all of them. Loading unnecessary dependencies will waste a considerable amount of physical memory due to the run-time overhead.

5.3.2 System performance

Generally speaking, using static libraries has better overall system performance, particularly in the application startup phase, since relocation, symbol lookup and resolution are processed at link-time. Using shared libraries may result in run-time performance penalties due to the following reasons:

1. A shared library and its dependencies must be dynamically mapped and parts of their contents must be decompressed and loaded by the run-time linker for the first time.

2. Relocation, symbol lookup and resolution of a shared library as well as its dependencies are very costly at run-time. A successful symbol lookup must match the whole string, comparing dozens of characters takes a lot of time. C++ mangling mechanism increases the lengths of symbol strings which in turn results in large dynamic symbol tables, decompressing and loading a large dynamic symbol table in physical memory is time consuming.

3. Shared libraries are usually compiled in position independent mode. References and calls to imported variables and functions are carried out indirectly through GOT and PLT entries. This will result in slower performance. According to Scott Bronson's statistics [10], calling a function in a shared library is 1%-20% slower than the static one in worst-cases. The variations of performance dependent on the GCC complier optimization level.

The above three performance penalties will not occur while using static libraries. However, in our case, these costs can be more or less offset by saving the decompression time once the operating system has already mapped and loaded a shared library in physical memory, and subsequent applications using the shared library are loading and executing. The more applications use a shared library, the less run-time performance costs they have to pay.

5.3.3 Compatibility and extendibility

In the case of using shared libraries, functions and variables are not statically bound to an application but are dynamically bound when the application is loaded. This permits applications to automatically inherit changes to the shared libraries, without recompiling or rebinding. This advantage can be utilized to make plug-in programs which can add new features and extended functionalities to the current applications. However, dynamically linked applications are dependent on compatible shared libraries. If a shared library is changed (for example, some interfaces of the shared library are changed or deleted) and becomes incompatible, this will lead to the applications no longer working. If a shared library is removed from the system, applications using that library will no longer work either. These problems would not happen, if applications are linked against static libraries, since all code is contained in the applications.

5.3.4 General proposals

Both the impacts on physical memory from file size and the large run-time overhead of shared libraries complicate the memory usage analysis dramatically. In addition, how shared code would be shared by applications and how many applications would share shared code will also affect the memory usage greatly. Therefore, it is barely possible to predict how static libraries and shared libraries will affect physical memory usage in reality. However, the following general proposals may be useful in the extreme cases.

1. If shared code is currently used by a single application, it is strongly recommended to put it into a static library. Except the cases that shared libraries are used as plug-ins which require frequently update.

2. If the size of resulting object files is quite large and the number of files sharing this

code is relative large. In addition to the above, if you are doubtless that most shared code will be shared by the majority of executable files, it it also recommended to compile the shared code to a shared library.

3. In a more general case, it would be a good idea to run the tools to determine which version of a library is more memory efficient.

5.4 How to remove unused material

It is wasteful to include unused functions and data in built object files. This unused material expands the size of object files, that will result in unnecessary relocation and memory waste. However, searching and eliminating unused material by hand is troublesome and time consuming. Fortunately, the link-editor provides this ability by passing the -gc-sections option to it, it will removed all unused sections. There is a problem with this method, since the sections such as .text and .data often contain more than one function and variable, the link-editor can not eliminate the entire sections unless all including functions or variables are useless. The solution to this problem is to separate each function and data into individual section. This section separation is achieved by using compiler options, i.e., the -ffunction-sections and -fdata-sections options. The -ffunction-sections and -fdata-sections options place each function or data into it's own section named .text.function_name or .data.data_name instead of a big .text or a big .data section. Armed with the -gc-sections, the ffunction-sections and the -fdata-sections options, the link-editor is able to track references of symbols and discard unused sections. Removed sections can be displayed by using –print-gc-sections option. After we applied this method on the Motorola executable files, the overall decrease in code size is 2% - 9%.

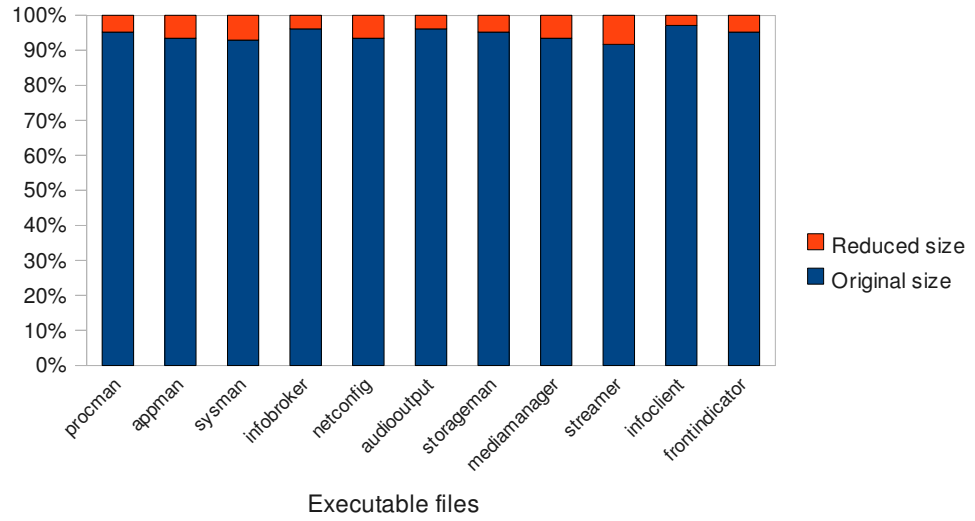

Figure 13. The proportion of the reduced file size to the original size.

The inclusion of unused library dependencies is also wasteful. Normally, the link-editor will consider all input shared libraries on the command line as dependencies of the output object file, regardless of whether the shared libraries are actually referenced by the built object or

not. Linking to these unused library dependencies will result in unnecessary loading and processing of shared libraries, which will affect run-time performance and memory usage considerably. This problem can become even worse when such unused shared library dependencies have their own dependencies, because, by default, the run-time linker will load and process any additional dependencies. To be more precise, the run-time linker will load library dependencies in a breadth-first traversal order which is starting with the libraries directly linked to the executable. Therefore, it is essential to exclude those unused dependencies, the link-editor provides this ability by passing the –as-needed option to it. The –as-needed option tells the link-editor to only records shared libraries which are actually referenced by the built object as dependencies.

5.5 Position independent code (PIC)

In the common case, code can be compiled in two different modes. By default, code is position dependent. Putting position dependent code into a shared library will cause the link-editor to generate a lot of relocation information, and cause the run-time linker to do a lot of processing at run-time. In addition, absolute virtual addresses of functions and variables will be encoded as a part of the instruction in the text segment. Since absolute virtual addresses are only known to the run-time linker, the text segment will be modified at run-time. This relocation of the text segment is referred to as text relocation. Due to text relocation, the modified text page can not be shared by other processes any more, each process requires its own private copy, this will compromise the shareability of the text segment.

Code can also be compiled in position independent mode, typically with the -fpic or -fPIC option. Position independent code does not contain absolute virtual addresses in the text segment. On the contrary, it contains addresses in the Global Offset Table (GOT) which resides in the data segment and is private to each process. Therefore, any modification of addresses in the GOT by the run-time linker will not compromise the shareability of the text segment. Position independent code enables the run-time linker to load shared libraries at any location of a process' virtual memory regions without modifying the text segment.

5.5.1 Differences between -fPIC and -fpic

The compiler can generate position independent code under either the -fPIC or the -fpic option. For some architectures, there is a limit of the size of GOT when compiling code with the -fpic option. For example, the limit number of GOT entries of SPARC is 2048, if the number of entries exceeds this limit, the link-editor will generate a fatal error. To overcome this error, we can recompile those source files with the -fPIC option. According to Ulrich Drepper's research [11], for some architectures, such as SPARC, there is a big difference between the -fPIC and the -fpic option with respect to instructions, code compiled under the -fPIC option requires two instructions (the compiler has to generate some instructions which can deal with any number of GOT entires) than the -fpic option when references to an imported variable or calls an imported function. Therefore, for such architectures, it is recommended to use the -fpic option for every source file and recompile those source files that exceed the size limit with the -fPIC option. However, the comparison of disassembled code of the Motorola shared libraries that compiled under the -fpic option with the -fPIC option indicates that there are not any differences at all, i.e., neither additional instructions are

needed nor GOT limit existing. Therefore, for the SH4 architecture, both the -fPIC and the -fpic options are applicable.

5.5.2 Relocation processing

In position dependent mode, relocation of symbols must be carried out by the run-time linker immediately when a shared library is loaded. On the contrary, in position independent mode, some relocation, typically calls to global functions, can be postponed until the functions are actually called by some code. The process of this delayed relocation is referred to as lazy binding. Lazy binding can speed up the start-up time of an application, because many functions will not be called when an application starts to execute.

In the following section I will simply explain how the relocation is carried out in position independent mode. There are two import data structures Procedure Linkage Table (PLT) and Global Offset Table (GOT). How these two tables are constructed is architecture-specific, however, the general principle is similar and applicable. The GOT holds the absolute virtual addresses of imported variables and functions. In position independent mode, the link-editor will create an entry for each imported global function in the PLT which resides in the shareable text segment. These entries make up the .plt section. The link-editor will also create a relocation entry in the .rela.plt section that tells the run-time linker which symbol is associated with the specified entry in the PLT. Code will call an imported global function through an associated entry in the PLT. As part of the first call to a PLT entry, control will be passed to the run-time linker. The run-time linker looks up the required function symbol and modifies the absolute virtual address of the associated entry in the GOT with the actual function address. The subsequent calls of the same PLT entry will not go through the run-time linker, instead go to the function directly. Furthermore, for each reference to an imported variable, the link-editor will create a relocation entry in the .rela.dyn section and an associated entry in the GOT as well. Unlike the relocation of function calls that can be lazy binding, the relocation of variables must be performed immediately before an application starts to run. The compiler will generate an instruction to load the absolute virtual address of a variable from the associated GOT entry, and then gets the actual value of the variable.

5.5.3 Cost of position independent code

From the above description, we can see that the most notable characteristic of position independent code is indirect references to variables and calls to functions. These indirect references and calls imply the cost of using position independent code, compared with position dependent code. We can summarize the cost as below:

1. For each call to an imported function, we need one PLT entry and one GOT entry.

2. For each reference to an imported variable, we need at least one extra load instruction and one GOT entry.

3. Position independent code is slightly slower when it calls an imported function or references to an imported variable. The code refers to such objects indirectly through the PLT and GOT. The specific amount of performance degradation mainly depends on the number of dynamic references to imported objects.

In sum, extra instructions and PLT entries indicate the incremental text segment size of a shared library, likewise, extra GOT entries indicate the incremental data segment size, therefore, the size of a shared library compiled in position independent mode should be larger than position dependent mode apparently. Is this conclusion correct? In order to verify this conclusion, several experiments were carried out on the Motorola shared libraries. Each experimental shared library was compiled with and without -fPIC option, thus, we obtained one position dependent version and one position independent version for each shared library. By comparing the size of two versions of shared libraries, unexpectedly, we observed that in most cases the size of position independent version was even smaller than the position dependent version. That subverts what we have concluded previously. In order to find out the reason, the size of each section of position independent and position dependent versions was compared. The sections with the most significant size changes were the .text section, the .plt section, the .got section, the .rela.dyn section and the .rela.plt section. Among these sections, two relocation associated sections (the .rela.dyn and the .rela.plt sections) caught our eyes. The .rela.plt sections consists of relocation records of dynamically linked functions which are called through PLT. The .rela.dyn section contains relocation records of dynamically linked data and functions which are not called through PLT. The size of the .rela.dyn section increased significantly in the position independent version compared with the position dependent version. The reason for this increase is that: in position independent mode, the relocation for each imported function will be performed only once and applied to the first instruction of a specific entry in PLT regardless of the number of this function called in the shared library. Likewise, the relocation for each imported data will also be performed only once. Therefore, there is only one relocation record with R_SH_JMP_SLOT relocation type for each imported function in the .rela.plt section and one relocation record with R_SH_GLOB_DAT relocation type for each imported data in the .rela.dyn section. Nevertheless, in position dependent mode, the relocation is performed as many times as the imported functions and data are referenced in the code. In addition, the relocation record for each imported function will be categorized into the .rela.dyn section, since no PLT entry is needed any more, moreover, the type of this relocation record will be changed to R_SH_DIR32. Thus, the specific number of relocation records for each imported function and data in the .rela.dyn section depends on the the number of times of these function and data called or referenced. Because each relocation record is 12 bytes and many imported function and data are referenced repeatedly, so the size of the .rela.dyn section will be expanded several times when a shared library compiled in position dependent mode. Sometimes, this increased amount is even larger than the total decreased amount from the text section, the .plt section and so on.

In conclusion, the size of a shared library compiled in position independent mode is not certainly larger than in position dependent mode, sometimes we can even benefit from position independent code in terms of size. Through the above analysis, there is an important fact revealed namely, that position dependent code generally requires much more relocation to be processed which will result in a considerable performance degradation in terms of application start-up time. Furthermore, there is also a moderate security risk in position dependent shared library. Because the text pages which involved in text relocation will be marked as writeable instead of read-only by the run-time linker temporally, so the attackers

can exploit this time slot and modify the code. All in all, it is strongly recommended to compile shared libraries in position independent mode.

5.5.4 Why not use PIC for all applications?

If position independent code has so many advantages, why not use it for all applications? As we mentioned previously, several experiments were performed to explore how position independent code would affect shared libraries in terms of size, we concluded that the change of size was not absolute. However, we also observed that there was an increase in size of each object file in position independent mode compared to position dependent mode. This implies that the cost of position independent code in terms of size will still exit in the object files which do not intend to compose a shared library. In other words, the size of executable files and static libraries which contain position independent code will be larger than position dependent code. For normally executable files, there is no text relocation needed, because all absolute virtual addresses are at the same location in a process' virtual memory space. Moreover, due to the cost of position independent code and size increase, compiling executable files in position independent mode will not bring us less start-up time and less memory usage by making text relocation unnecessary, instead it will result in a considerable performance penalty due to indirect references. Therefore, it is not recommended to apply position independent mode to executable files. For static libraries, the situation becomes more complicated, we must deal with them case by case, although a static library is a simple archive of object files. According to the intention of a static library, we can categorize our recommendations into there cases:

1. If a static library is intended to be used by executable files exclusively, it is recommended to build it in position dependent mode.

2. If a static library is intended to be used by shared libraries exclusively, it is recommended to build it in position independent mode.

3. If a static library is intended for both shared libraries and executable files, it is recommended to build two versions. A position independent version will be used by shared libraries, another position dependent version will be used by executable files. Since static libraries do not need to be loaded into set-top box, no extra physical memory is required to perform this method.

6 Summary

In this thesis, we have gained knowledge of the ELF format for object files in Linux. The linking and loading process of ELF object files along with Linux virtual memory management have also been studied. In fact, the most time-consuming and tedious part of this work is the understanding of the background theories. Perhaps due to the continuously evolving of free software contributed by decentralized programmers all over the world, the Linux documentation is quite poor.

6.1 Conclusion

During our implementation phase, we observed that adopting a shared library would impose a considerable amount of run-time memory footprint overhead and performance overhead. The main cause of this overhead is due to the relocation and load-time symbol resolution of a shared library. Therefore, eliminating unnecessary dependencies is very important. Besides, we also found out that the components of shared code and how these components were shared by applications as well as the number of applications sharing the shared code and the impacts on the file size from static and shared libraries were the determinant factors for the destination of shared code. All these factors complicate the analysis dramatically in reality. Therefore, it is barely possible to predict how static libraries and shared libraries will affect physical memory usage in practice, except some extreme cases. Although a generic method that determines when to choose one type of libraries over the other in advance is not available, converting an existing static or shared library and running our tools could still give us the sufficient evidence of whether keeping the current type of a library or not. In addition to the above, we also concluded the benefits of compiling shared libraries to position independent code mode outweighed the performance penalties it brought.

6.2 Future work

Due to time constraints, currently, our tools are limited to the 1920-9C IP-STB, it is better to extend the existing tools to be able to be applied on other series of IP-STBs. Since we only focused on the Motorola's proprietary code, the third party code was not considered. It would be interesting to investigate third party code in future, since the third party focuses more on feature completeness than embedded usage. Besides, all the performance analysis are theoretical, if a method of measuring the performance variations will be proposed in future, it will give us a deep insight into the theoretical performance costs in practice. There are still several interesting topics about linker optimization options that have not been covered, such as the -O option which optimizes the hash table size, -z now and -z relro options which are related to the relocation. They are also worth being investigated in the future.

7 References

[1] *VIP1920-9C & VIP1970-9C Specification Sheet 2008.* Motorola, Linköping.

[2] *Kreatel IP-STB Developers Documentation.* Motorola, Linköping.

[3] *BusyBox* www.busybox.net [accessed 11/07/2010].

[4] Daniel Robbins (2001). *Advanced filesystem implementor's guide*. http://www-128.ibm.com/developerworks/library/l-fs3.html [accessed 15/07/2010].

[5] Christoph Rohland (2001). *Tmpfs*. linux-2.6.31.12/Documentation/filesystems/tmpfs.txt

[6] Phillip Lougher. *Squashfs.* http://squashfs.sourceforge.net/ [accessed 15/07/2010]

[7] *Executable and Linking Format Specification* (1995). TIS Committee.

[8] Daniel P. Bovet, Marco Cesati (2005). *Understanding the Linux Kernel, 3*rd *edition.* United States of America: O'Reilly Media.

[9] Linus Torvalds (2008). *Linux Kernel ChangeLog.* *http://kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.25* [accessed 17/07/2010]

[10] Scott Bronson (2004). *Peformance testing shared vs. static libs*. http://gcc.gnu.org/ml/gcc/2004-06/msg01956.html [accessed 18/08/2010]

[11] Ulrich Drepper (2006). *How To Write Shared Libraries*. Red Hat, Inc.

8 Appendix A

The code segment of the modified “Pss” patch.

```
@@ -12,7 +12,7 @@
                        seq_printf(m,
                                   "Size:           %8lu kB\n"
                                   "Rss:            %8lu kB\n"
-+                                 "Pss:            %8lu kB\n"
++                                 "Pss:            %8lu B\n"
                                   "Shared_Clean:   %8lu kB\n"
                                   "Shared_Dirty:   %8lu kB\n"
                                   "Private_Clean:  %8lu kB\n"
@@ -20,7 +20,7 @@
                                   "Referenced:     %8lu kB\n",
                                   (vma->vm_end - vma->vm_start) >> 10,
                                   mss->resident >> 10,
-+                                  (unsigned long)(mss->pss >> 22),
++                                  (unsigned long)(mss->pss >> 12),
                                   mss->shared_clean  >> 10,
                                   mss->shared_dirty  >> 10,
                                   mss->private_clean >> 10,
```

9 Appendix B

Memory usage comparison reports of converting static libraries to shared libraries.

Converting libcutils.a to libcutils.so

Process	Vss(kB)	Rss(kB)	Pss(B)
procman	4732	2020	533844
	4800	2028	536574
halserver_7109	127324	33716	2654942
	127324	33716	2654942
sysman	6092	2596	653095
	6092	2596	653095
appman	6116	2368	582219
	6116	2368	582219
infobroker	5928	2416	640476
	5928	2416	640476
tdiserver	35036	31324	847015
	35036	31324	847015
netconfig	4612	1840	362444
	4612	1840	362444
audiooutput	6104	2356	567033
	6104	2356	567033
storageman	5668	2300	627744
	5668	2300	627744

mediamanager	7368	2504	803121
	7444	2524	818139
streamer	13416	4588	2399803
	13488	4596	2402533
infoclient	6044	2100	438996
	6044	2100	438996
frontindicator	6080	2296	525513
	6080	2296	525513
ekioh	24736	12356	6949835
	24736	12356	6949835

sum_pss_orig - sum_pss_new = -19.998 kB
tmpfs_size_orig - tmpfs_size_new = -4 kB
we lost 23.998 kB memory

Converting libproc.a to libproc.so

Process	Vss(kB)	Rss(kB)	Pss(B)
procman	4732	2020	533844
	4824	2044	537354
halserver_7109	127324	33716	2654942
	127412	33728	2646164
sysman	6092	2596	653095
	6184	2620	656605
appman	6116	2368	582219
	6116	2368	582219

infobroker	5928	2416	640476
	6028	2448	652178
tdiserver	35036	31324	847015
	35036	31324	847015
netconfig	4612	1840	362444
	4708	1868	370050
audiooutput	6104	2356	567033
	6104	2356	567033
storageman	5668	2300	627744
	5760	2328	635350
mediamanager	7368	2504	803121
	7444	2524	818139
streamer	13416	4588	2399803
	13512	4616	2407409
infoclient	6044	2100	438996
	6044	2100	438996
frontindicator	6080	2296	525513
	6080	2296	525513
ekioh	24736	12356	6949835
	24736	12356	6949835

sum_pss_orig - sum_pss_new = -31.994 kB
tmpfs_size_orig - tmpfs_size_new = -16 kB
we lost 47.994 kB memory

Memory usage comparison reports of converting shared libraries to static libraries.

Converting libhalcppclient.so to libhalcppclient.a

Process	Vss(kB)	Rss(kB)	Pss(B)
procman	4732	2020	533844
	4732	2020	533844
halserver_7109	127324	33716	2654942
	127324	33716	2656141
sysman	6092	2596	653095
	5924	2496	678188
appman	6116	2368	582219
	4960	2048	489907
infobroker	5928	2416	640476
	5928	2416	641675
tdiserver	35036	31324	847015
	35036	31324	848078
netconfig	4612	1840	362444
	4612	1840	362444
audiooutput	6104	2356	567033
	5924	2240	573557
storageman	5668	2300	627744
	5668	2300	628211
mediamanager	7368	2504	803121

	7444	2524	804184
streamer	13416	4588	2399803
	13120	4424	2357311
infoclient	6044	2100	438996
	5876	1992	445521
frontindicator	6080	2296	525513
	5756	2140	489030
ekioh	24736	12356	6949835
	24736	12356	6951034

sum_pss_orig - sum_pss_new = 123.980 kB
tmpfs_size_orig - tmpfs_size_new = -56 kB
we saved 67.980 kB memory

Converting libsubtitlesource.so to libsubtitlesource.a

Process	Vss(kB)	Rss(kB)	Pss(B)
procman	4732	2020	533844
	4732	2020	533844
halserver_7109	127324	33716	2654942
	127324	33716	2654942
sysman	6092	2596	653095
	6092	2596	653095
appman	6116	2368	582219
	6116	2368	582219

infobroker	5928	2416	640476
	5928	2416	640476
tdiserver	35036	31324	847015
	35036	31324	847015
netconfig	4612	1840	362444
	4612	1840	362444
audiooutput	6104	2356	567033
	6104	2356	567033
storageman	5668	2300	627744
	5668	2300	627744
mediamanager	7368	2504	803121
	7368	2524	803121
streamer	13416	4588	2399803
	13416	4588	2399803
infoclient	6044	2100	438996
	6044	2100	438996
frontindicator	6080	2296	525513
	6080	2296	525513
ekioh	24736	12356	6949835
	24736	12356	6949835

sum_pss_orig - sum_pss_new = 24.000 kB
tmpfs_size_orig - tmpfs_size_new = 4 kB
we saved 28.000 kB memory