Networked Multimedia Embedded Linux

Case Study

Version 1.1

11/30/2001

1 Overview 5
1.1 Case Study Goal 5
1.2 Hardware 5
1.3 Software 5
2 Software Reference 7
3 Software Overview 9
3.1 Linux 9
3.2 Microwindows + FLNX 9
3.3 ViewML 9
3.4 glibc 10
3.5 GStreamer 10
3.6 rrload 10
4 Porting GStreamer 12
4.1 Cross Compiling 12
4.2 Packaging 14
4.3 Pthreads 15
4.4 Base Libraries 15
4.5 Other Subsystems 16
4.6 Compilation Options 17
4.7 Conditional Compilation 17
4.8 Code Changes 19
4.9 TaskBridge DSP Support 20
5 Linux Kernel Improvements 22
5.1 Starting Point 22
5.2 xconfig – Feature Inclusion Configuration Tool 23
5.3 Static Buffer and Array Size Analysis 23
5.4 Dynamic Memory Usage Analysis 25
5.5 File System 29
5.6 Networking 31
5.7 Network Tuning 33
5.8 Target Device File System Image 33
5.9 Subsystem Replacement 36
6 glibc Improvements 41
6.1 Alternate C Libraries 41
6.2 Removing Unused Library Functions 42
7 Audio Playback Performance Analysis 45
7.1 ARM Idle CPU Utilization 45
7.2 ARM MP3 Playback CPU Utilization 46
7.3 C55 DSP MP3 Playback CPU Utilization 47
8 Summary 49
9 Appendix A - Software Documentation 52
9.1 Linux Utility Commands 52
10 Appendix B - GStreamer Documentation 54
10.1 GstElement 54
10.2 GstPad 54
10.3 GstBin 55
10.4 GstScheduler 55
10.5 GstBuffer 55
10.6 GstCaps/GstProps 56
10.7 GstPlugin 57

11 Appendix C - Case Study Starting Configuration....58
12 Acknowledgements....60
12.1 libwww Copyright Notice....60
12.2 W3C® SOFTWARE NOTICE AND LICENSE....60
12.3 MPG123 License....60

Figures

Figure 1: High-level software diagram....5
Figure 2: Kernel subcomponents....5
Figure 3: Virtual File System Example....37
Figure 4: Linux memory management components....38
Figure 1: Program simplest.c used for maximum libc library optimization....42
Figure 2: List of unresolved externals in compiled version of simplest.c....42
Figure 5: ARM CPU utilization when idle....45
Figure 6: GStreamer Example Pipeline....54

Tables

Table 1: Software component sources 7
Table 2: Software library sources 8
Table 3: Tool sources 8
Table 4: Development workstation directory layout 12
Table 5: Target device directory layout 13
Table 6: GStreamer libraries with various compacting options 17
Table 7: GStreamer conditional compile directives 18
Table 8: GStreamer conditional compile memory reduction 19
Table 9: Memory reduction via `xconfig` 23
Table 10: Linux kernel code changes for static variable / array memory reduction 24
Table 11: Memory reduction via static variable / array related code changes 24
Table 12: bss reduction via static variable / array related code changes 24
Table 13: Memory reduction via `xconfig` and static variable /array code changes 25
Table 14: Summary of memory reduction via dynamic memory related code changes 29
Table 15: Cache sizes versus installed RAM memory 30
Table 16: Case study file system memory reduction 30
Table 17: Case study versus uClinux 2.0.38 file system comparison 31
Table 18: Size of case study kernel with various networking capabilities enabled 31
Table 19: Size of 2.4.10 uClinux kernel with various networking capabilities enabled 32
Table 20: Size of uClinux 2.0.38 kernel with various networking capabilities enabled 32
Table 21: Comparison of various kernels with various networking capabilities enabled 32
Table 22: Unsolved file system variables / function calls 33
Table 23: CRAMFS detailed file size reduction 35
Table 24: CRAMFS file system image size reduction 36
Table 25: libc size comparison 41
Table 8: Library Optimizer results 43
Table 26: New applications working with a functionally reduced library 44
Table 27: CPU utilization for MPG123 constant generation 46
Table 28: CPU utilization for MPG123 MP3 decoder 47
Table 29: CPU utilization for Imagine Technologies C55 MP3 DSP decoder 47
Table 30:Imagine Technology C55 MP3 Decoder Specifications 48
Table 31: Memory footprint reduction options summary 51

Listings

Listing 1: Kernel starting size for `xconfig` 22
Listing 2: Largest statically allocated variables 23
Listing 3: Slab allocator results 26
Listing 4: Kernel memory allocator results 27
Listing 5: Dynamic memory allocation starting point 27
Listing 6: Available memory starting point 28
Listing 7: Dynamic memory information results 28
Listing 8: Memory available after disabling swap daemon 29
Listing 9: Uncompress file system image size 34
Listing 10: CRAMFS compress file system image size 35

1 Overview

1.1 Case Study Goal

This case study focuses on the engineering challenges relating to RAM and ROM/flash memory usage in embedded devices – specifically, the changes typically required when re-implementing desktop-targeted open source projects in an embedded device. The specific components being studied include the Linux operating system and the GStreamer streaming media framework.

1.2 Hardware

The embedded device hardware platform used for this technology case study is the TI OMAP 1510 P1 evaluation module (EVM). The features of the P1 EVM include:

- ARM9 general purpose processor
- C55 digital signal processor
- Audio output via an I2S audio codec with power amplifier and speaker connection
- LCD display with touch screen
- 10T networking

There are many other features on the P1 EVM, however they are outside the scope of this case study.

In addition to the P1 EVM, a network connected development workstation running Redhat 7.1 / Linux 2.4.2 is also used for cross development.

1.3 Software

The software for the P1 EVM is shown in the software diagram in figure 1.

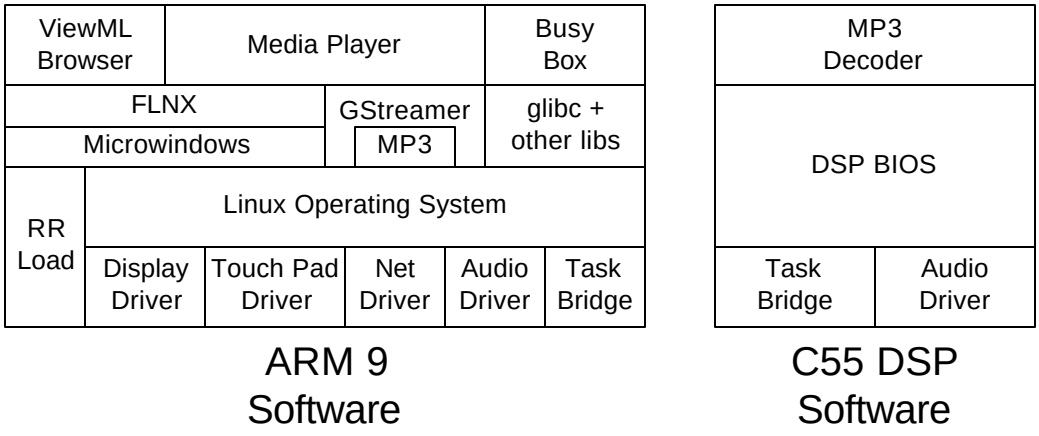

Figure 1: High-level software diagram

The subcomponents in the Linux kernel are shown in figure 2.

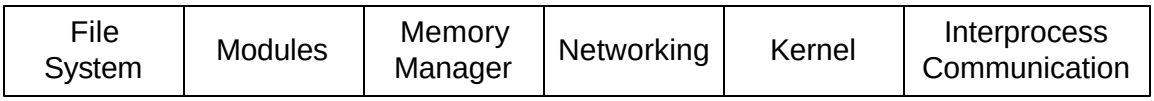

Figure 2: Kernel subcomponents

In addition to the software running on the P1 EVM, we also utilize a collection of software development compilers, assemblers, linkers, and other development utilities (referred to as a tool chain). Details about each software component are provided below in the Software Reference section.

2 Software Reference

The software used for this case study came from many sources. Table 1 lists each component, its version and its source.

Software Component	Version	License	Source
Display Driver	0.9	Proprietary	RidgeRun
Touch Pad Driver	0.9	Proprietary	RidgeRun
Net Driver	2.4.0 test #12	GPL	Part of Linux kernel
TaskBridge Linux Kernel	0.9 2.4.0 test #12	Proprietary GPL and LGPL	RidgeRun http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.0.tar.gz
Microwindows	0.89pre7	MPL or GPL	http://www.microwindows.org/
FLNX	0.16	GPL	ftp://microwindows.censoft.com/pub/microwindows/flnx/
GStreamer	0.2.1+	LGPL	http://www.gstreamer.net/
glibc	2.1.3	LGPL	ftp://ftp.gnu.org/gnu/glibc/glibc-2.1.3.tar.gz
ViewML Browser	CVS 9/25/01	GPL	http://www.viewml.com/
Media Player	0.2.1+		Part of GStreamer
BusyBox RR Load	0.60.1 0.9	GPL Proprietary	ftp://oss.lineo.com/busybox/
ARM9 Audio Driver	0.9	GPL	http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/armlinux/linux/drivers/sound/ (sa1100-audio.c, sa1100-audio.h, and sa1100-uda1341.c) modified to work on OMAP 1510 P1 EVM
TaskBridge C55 DSP MP3 Decoder	0.9 0.9	Proprietary Proprietary	RidgeRun Imagine Technology, LLC http://www.imaginetechnology.net/
mad ARM9 MP3 Decode	0.13	GPL	http://sourceforge.net/projects/mad/
mpg123 ARM9 MP3 Decoder	0.59r	mpg123 custom	http://www.mpg123.de/

Table 1: Software component sources

In addition to the main software components listed above, various libraries are also used, as listed in Table 2.

Software Component	Version	License	Source
imlib	1.9.10	GPL	For ViewML and Microwindows http://freshmeat.net/projects/imlib/
libwww	5.2.8	W3C® software notice and license	For ViewML http://www.w3.org/Library/User/History.html
jpeg	6b	libjpg custom	For Microwindows http://www.ijg.org/files/ This software is based in part on the work of the Independent JPEG Group
zlib	1.1.3	zlib custom	For Microwindows and libpng http://www.gzip.org/zlib/ Copyright (C) 1995-1998 Jean-loup Gailly and Mark Adler
libpng	1.2.0	libpng custom	For Microwindows http://www.libpng.org/pub/png/libpng.html
glib gobject gmodule gthread	1.3.10	LGPL	For GStreamer ftp://ftp.gtk.org/pub/gtk/v1.3/

Table 2: Software library sources

Various tools were used as part of this case study, as listed in Table 3:

Package	**Tools**	**Version**	**Source**
gcc	gcc	2.95.2	http://www.gnu.org/software/gcc/gcc-2.95/gcc-2.95.2.html
gcc arm patches		10/22/99	ftp://ftp.netwinder.org/users/u/urnaik/gcc-2.95.2-diff-991022.gz
binutils	addr2line, ar, c++filt, demangle, gas, gprof, ld, nlmconv, nm, objcopy, objdump, ranlib, readelf, size, strings, strip, and windres.	2.10	ftp://ftp.gnu.org/gnu/binutils/binutils-2.10.tar.gz
Library optimizer	libraryopt	1.0.1	http://sourceforge.net/projects/libraryopt/

Table 3: Tool sources

3 Software Overview

For this case study, the focus centered around memory issues of Linux solutions in embedded devices. The two main areas of investigation were porting the GStreamer open source multimedia desktop software technology into the embedded space, along with improvements to Linux kernel memory usage.

3.1 Linux

The Linux kernel was designed from its inception in high level language such that it could run on different families of microprocessors provided that they were supported by a common cross-development environment, including compiler, assembler, linker, and debugger. Linux works in embedded devices if a Linux port to the embedded processor used in the target device exists. Linux device drivers for embedded device I/O interfaces and peripherals must be ported or written. Tools to download code to the target device are required. Most of the necessary software pieces are available on the Internet, but pulling together the entire package is time consuming – taking an estimated 3 to 6 engineering months to integrate into an effective development environment. Finding a supported Linux distribution for the target embedded processor is more efficient. For this case study, the RidgeRun OMAP 1510 DSPLinux BSP, version 1.0 was used, with some of the measurements made using a pre-release version of the BSP.

3.2 Microwindows + FLNX

The Linux kernel does not include a graphical user interface (GUI). Instead, a separate library is used. For this case study Microwindows is used for the GUI and FLNX provides the widget set.

Microwindows is an open source project aimed at bringing the features of modern graphical windowing environments to smaller devices and platforms. Microwindows applications can be built and tested on a Linux development workstation, as well as cross-compiled for the target device. Microwindows consists of the Win32/WinCE API and the Xlib-like API implementation known as Nano-X. Microwindows uses the Linux frame buffer to display output and the touch pad driver to receive user input.

On top of Microwindows is FLNX, a version of the FLTK (fast light tool kit) application development environment modified to target Nano-X rather than X. FLNX provides widget set support; including buttons, dialog boxes, text boxes, and widgets for the input of numeric values. The case study did not attempt

3.3 ViewML

The ViewML browser is an open source project aimed at producing a small-memory footprint, high-quality web browser for the embedded Linux market. The ViewML browser is based on KDE's kfm (kde file manager), and FLTK. The kfm HTML widget has extensive support of HTML v3.2. The kfm HTML widget displays most of today's web site contents without error. Kfm is written in C++, and originally required the Qt widget set for various user interface controls. The standard Qt widget set, however, is too large for most embedded requirements, so ViewML uses a translation layer that allows the kfm HTML widget to use FLTK. ViewML supports a customizable user interface. The display engine also uses FLTK for forms support.

3.4 glibc

The glibc library supports the standard C language APIs. Some functions are handled completely within the library, such as the string functions `strcp()`, `strcmp()`, etc. Other functions invoke the Linux kernel, such as file system functions `open()`, `read()`, `write()`, and `close()`. In the x86 desktop PC environment, the size of the current version of glibc library is about the same size as the size of the current version of the Linux kernel itself. Detailed code size information is provided latter in the case study.

3.5 GStreamer

GStreamer is a framework for creating streaming media applications. GStreamer's development framework makes it possible to write any streaming multimedia application. The framework includes several components to build a full featured media player capable of playing common media encodings including MPEG1, MPEG2, AVI, MP3, WAV, and AU. GStreamer, however, is much more than just another media player. Pluggable components also make it possible to write a full-fledged video or audio editing application. For embedded devices, pluggable components can be developed for special effects, like reverberation or parametric equalization.

GStreamer allows the construction of graphs of media-handling components, ranging from simple MP3 playback to complex audio (mixing) and video (non-linear editing) processing. Applications can take advantage of advances in codec and filter technology transparently. Developers can add new codecs and filters by writing a simple plugin with a clean, generic interface. GStreamer is released under the LGPL, with many of the included plugins retaining the license of the code from which they were derived, usually GPL or BSD.

The framework is based on plug-ins that provide codec and other functionality. The plug-ins can be connected and arranged in a pipeline. This pipeline defines the data flow. Applications sit on top of the GStreamer framework to use the supplied multimedia functionality without having to know about the details. For example, an application written with a GStreamer framework containing an MP3 codec will automatically be usable for playing AAC streamers once an AAC codec plug-in is installed.

For the desktop PC, the plug-ins are executed by the PC processor, usually an x86 microprocessor. For the case study, the GStreamer framework was modified to allow plug-in execution by the DSP.

More information about GStreamer can be found at http://www.gstreamer.net/.

3.6 rrload

The rrload boot loader was developed by RidgeRun Inc, and is shipped as a component within the DSPLinux SDK distribution. It is tailored specifically to manage the loading, storing, and invoking of a Linux kernel and root file system. In normal operation, the boot loader resides in flash (at the reset vector) and is the first program run on power up, and unless intercepted, rrload will typically transfer control to the stored system (e.g. kernel + file system). Additionally, the boot loader will relocate either the kernel and/or the file system to SDRAM if necessary prior to transferring control. This behavior happens automatically, responding to a previously configured rrload with a default boot command of "boot_auto". This is the typical command stored along with the user's other persistent boot loader settings, which among other items, can include extra kernel command line arguments.

The rrload tool can also transfer control to a Linux kernel compiled for execute-in-place (XIP) operation. Execute-in-place has the advantage of reducing the RAM needed (but typically requiring more ROM/flash since the executables can not be stored compressed). Execute-in-place is typically not used if XIP is significantly slower than executing in RAM due to the difference in memory access speeds for flash compared to RAM. For this case study, XIP was not used.

If the boot loader is not configured with a default boot command, or if the boot process is intercepted, rrload will simply present its user interface and wait for user input. The boot loader offers both a menu UI, as well as a command line UI that the user may easily toggle between. The ability to download a new kernel and/or file system to either RAM or flash is available via the UI. The boot loader supports two download formats. The first format is Motorola `srec` format. The second format is called `rrbin` and consists of a three line tag appended to the front of a binary image. The tag consists of the load address, the start address, and the binary image length. The rrload boot loader can accept these download formats over a variety of board I/O ports such as serial, parallel and Ethernet. The user interface communicates over the board's serial line with a host terminal session such as minicom, HyperTerminal, etc. Holding the [Enter] key down within the host terminal session while simultaneously applying power to the board will insure that any previously stored default boot command is temporarily intercepted, forcing the boot loader to present its user interface.

4 Porting GStreamer

Like most open source projects, GStreamer is focused on functionality over minimizing footprint or maximize performance. It was originally targeted for x86 desktop PCs with features appropriate for desktop usage. Modifying GStreamer to function in the target embedded device follows a series of steps likely to form a pattern used by other open source projects transitioning to the embedded space. Those steps, and the challenges are described below.

A functional description of GStreamer can be found in *Appendix B - GStreamer Documentation* on page 54.

4.1 Cross Compiling

Cross-compiling is the process of building software for a target processor architecture on a development workstation that is based upon a different host processor architecture. The software for the ARM target device is cross-compiled using an x86-based development workstation. The GNU tools enable cross compiling – however, adapting for embedded specific requirements is necessary.

The collection of compilers, assemblers, linkers, and other development utilities is referred to as a tool chain. The tool chain used for cross development is typically installed on the development workstation in an auxiliary location such as /opt/arm-linux, to avoid interactions with the native tool chain on the development workstation. Conventionally, the environment variable $prefix holds the directory of the tool chain to use. When cross developing, $prefix is set to /opt/arm-linux. The glibc C library is installed in the $prefix directory in order to build programs that use the C library available on the target device. In order to find glibc, the tool chain has include and library search paths hard coded into the various tools, requiring that we install the libraries into $prefix/lib directory. One of the most common mistakes in cross development is to build and test the program on the development workstation and then attempt to cross compile for the target device. In the cross-development linking stage, all the library routines available on the development workstation, but not available on the target device, turn up as unresolved externals. It often takes a significant amount of rework to get the program working without depending on libraries only available on the development workstation.

On the target device, the C libraries are installed in their 'usual' location, /lib, allowing applications to locate them during the normal boot sequence and. Other libraries are normally installed in /usr/lib (thus the target device system's $prefix is '/usr').

Directory	Contents
/usr/bin/	Native compile tools
/usr/lib/	Native shared libraries
/usr/include/	Native header files
/opt/DSPLinux/TI925DC_EVM/crossdev/bin/	Compile tools for target device
/opt/DSPLinux/TI925DC_EVM/crossdev/arm-linux/usr/lib/	Target device shared libraries
/opt/DSPLinux/TI925DC_EVM/crossdev/arm-linux/usr/include/	Target device header files
…/fs/	Entire file system image for the target device

Table 4: Development workstation directory layout

The elipse (…) indicates the developer is free to put the target device file system in any directory on the development workstation they choose. The target device related directories shown above are the default locations supported by the RidgeRun TI OMAP 1510 P1 EVM BSP. The files under …/fs/ in Table 4 are shown in Table 5.

Directory	Contents
/bin/, /sbin/, /usr/bin/, /usr/sbin/, /usr/local/bin/	Target device applications
/lib/, /usr/lib/	Target device shared libraries

Table 5: Target device directory layout

These other libraries are built in such a way as to make them available for cross-compiling. The build system used for many open-source projects, called autoconf/automake, allowing the $prefix path to be set both at configuration time and at installation time. This enables libraries, scripts, and applications for the target device to have paths relative to the installed system, but be installed into a different $prefix directory on the target device.

4.1.1 Autoconf'd Cross-Compilation

The GStreamer build system is based, like most open-source projects, on GNU's autoconf, automake, and libtool. These make the build process quite a bit simpler to deal with in most cases. There autoconf tool contains a number of complexities that make larger projects more difficult to manage, but these are being solved, in many cases by patches originating from the GStreamer development team.

When configuring a package to be built on the development workstation, the autoconf configure script runs a large number of tests to check the compile-time and run-time environment. This includes checking for headers and libraries, checking the sizes and values of certain system-supplied symbols, and various other things designed to make the package portable across a large number of targets.

When cross-compiling, the build system looks at the target device's environment, rather than the development workstation's native tool chain and libraries. The tool chain itself aids significantly in this process, because the default include and library search paths simply do not include the development workstation native paths. Any library relied upon while cross-compiling must be installed into the appropriate directory used by the tool chain for the target device. Making the developer aware of available libraries on the target device generally eliminates surprises about using libraries only found on the development workstation. Table 4 shows the directory layout.

The autoconf tool was designed under the assumption that the processor architecture of the target device is the same as the development workstation processor architecture. This assumption creates a problem when the autoconf configuration process needs to check the size or value of a given symbol provided by the build environment. Checking a symbol size or value is done under autoconf with AC_TRY_RUN, which attempts to run a program that will presumably output the result of the test in question. However, a cross-compiler produces binaries that cannot be run on the development workstation. Having an automated cross-compiler build process is difficult (to impossible) when a step requires a program to run on the target device. To maintain an automated build process, the GStreamer team examines each autoconf test run and modifies the GStreamer code that causes the test to be required in the first place. However, removing all platform dependent tests was not completely successful. To meet the case study deadline, GStreamer and associated libraries were built on an ARM9 NetWinder hardware platform to allow autoconf to run on the same processor as the used in the

target device. As the GStreamer team works through the remaining issues, a fully cross-development build system will be used.

4.1.2 Glib

Before GStreamer can be built, the Glib library must be compiled. Glib is used by GStreamer to provide common portability constructs, data types, and most importantly the object model. Because of its nature as a portability layer, the Glib configuration has a large number of checks that look into the build environment.

Most of the autoconf checks that require running a program to determine answers to test questions are either caused by a poorly designed macro supplied with Glib (to check the size of a type), or cases where the test program is run unnecessarily.

Some are extremely difficult to solve, however. The first challenge is determining the system capability to support pthreads. The configure script checks what thread libraries exist, and then performs a series of tests to ensure the chosen thread library (pthreads on Linux) function as desired. One check actually determines the contents of the pthread mutex initializer. For cross-compiling, this can be handled by simply including the normal PTHREAD_MUTEX_INITIALIZER symbol in the constructed header, though there may be issues with this solution on some processor architectures.

4.2 Packaging

Packaging is the process of taking a group of files owned by a certain program and putting them into a single archive file that can be installed on a system as a single piece. Packages allows for clean removal and upgrading of installed programs and simplifying initial installation.

The most common packager is RPM Package Manager. It uses a build system based on a SPEC file, which contains information about the package including name, version, dependencies, and an abstract, as well as the instructions for building and packaging the program.

One key note regarding RPM is its use of "BuildRoot"; a temporary directory into which the program is installed before being turned into a binary package. BuildRoot is used to avoid over writing the installed program while building a new version of an existing package. It takes advantage of autoconf's ability to use a different prefix for configuration and install time. The configuration prefix is set to /usr, typically, whereas the install prefix is set to $buildroot/$prefix.

Another feature of RPM is the ability to create relocatable packages. These are packages in which all the files (with certain exceptions) all start with the $prefix path. By supplying a $prefix directory, the user can control the package installation directory. All the files are then relocated into the $prefix location.

RPMs for development libraries are packaged typically into two distinct binary packages: runtime and development. The runtime package contains the dynamic library and required configuration and support files, whereas the development package contains the static version of the library (if any) and the headers required to build against that library.

For a cross-compilation environment, both the development and runtime packages are installed into the tool chain directory that contains the cross-compiler and C library for the target device. Controlling the installation location can be accomplished using the RPM relocatable package feature, which will move the target device files in /opt/arm-linux directory instead of the /usr/lib

directory, thus enabling the cross-compiler to find the library and header files. The same package also can be installed into the target device file system image (described in the Target Device File System Image section, page 33).

4.3 Pthreads

Most multiprocessor machines use threads to provide a convenient programming model. Linux takes advantage of multiprocessors using its symmetric multiprocessor (SMP) technology. The threads operate independently, but can synchronize using certain primitives (mutexes, condition variables, etc.). On a symmetric multiprocessor machine, the threads can run on separate processors simultaneously. The key feature is the sharing of memory space between multiple threads in the same process. On a dual-core system like the case study target device, threads would be a significant benefit, because allowing the use of the same programming model. However, Linux doesn't have a general solution for supporting asymmetric multiprocessor hardware.

Thread support on a dual-core system requires two elements. The first element is the ability to manage threads on the DSP, which implies a small scheduler running on the DSP and the ability to coordinate threads between the GPP and DSP. The second element is the ability for threads belonging to the same process to have a shared memory space. A GPP/DSP shared memory space requires a DSP MMU that can map memory in the same manner as the GPP MMU. The DSP MMU will need to be programmed and reprogrammed independent of the ARM MMU, but still allow access to the same pages as a thread in the same process running on the ARM. Synchronization primitives must be provided by way of inter-CPU interrupts, and/or atomic memory operations.

The real challenge in supporting threads on the DSP is the inability to run GPP code on the DSP. Another code set is required for the DSP. Properly dealing with the DSP code requires a set of tools to link the two kinds of binaries (ARM and DSP) together, as well as a loader that can work with the threading subsystem.

4.3.1 Impact on GStreamer

Because GStreamer has a thread container, i.e. one that causes all of its children to run in a different thread instead of the main pipeline context, it's a fairly straightforward process to extend GStreamer to use these kinds of pthreads. However, since they are not supported on the target device, another approach is needed.

A small GStreamer-compatible core could be created specifically for the DSP. The main issue is dealing with the different kinds of code. A GStreamer plugin subsystem modification supporting multiple code binaries would be needed. The case study investigated various approaches to solving the challenge of porting GStreamer, which relies on threads, to a dual-core environment where the DSP doesn't support threads. A different solution than the one described here was used. See the *DSP Support* section (page 20) for details.

4.4 Base Libraries

Although GStreamer is a library, it uses other libraries as well. The first hurdle in reducing the size of GStreamer was the fact that the initial implementation was done on top of the Gtk+ library, which is a GUI toolkit for X (and other environments). Gtk+ provides a well-designed object model written in C, which simplified the GStreamer design. However, it meant that any application built with GStreamer also was built with Gtk+, which in turn required an active X server (which is notoriously large in size).

The solution came a year later when the Gtk+ team moved the object model into the lower-level library, Glib, calling it GObject. This removes GStreamer's need to use the Gtk+ library entirely, along with the need for an X server. Although changing GStreamer to use GObject was straightforward, the transition was difficult because implementing GObject was challenging. The port was initially done using a GObject shim that converts the GObject-based code at compile-time to use Gtk+. When the GObject was done, the shim was removed. Removing the dependence on Gtk+ removed around a megabyte of code from the x86 version, probably more from the ARM equivalent. Refer to Table 8 (page 7) for a full description of the total memory footprint reduction.

4.5 Other Subsystems

In addition to the above subsystems, the entire DEBUG/INFO system can be compiled out. The DEBUG/INFO system normally gives significant debugging information in a very readable form. ANSI text colors show the category of each debug message, as well as the pthread and cothread ID. Each debug category can be turned on and off separately. During normal compilation only the INFO messages are compiled in. Development compilations also enables the DEBUG messages. For an embedded system, debug/info messages are disabled once debugging is complete.

4.6 Compilation Options

There are several options that can be given to the compiler to produce smaller code. The first and most obvious of these is '-Os', which tells the compiler to optimize for size. The original GStreamer makefile did not use complier options to minimize memory usage. Unneeded symbols can be stripped from the libraries (using the `strip` utility). Libraries that support disabling asserts and memory pools can be further reduced (using the conditional compile directives `G_DISABLE_ASSERT` and `G_DISABLE_MEM_POOLS`). These savings are show below.

Library	size	Stripped Symbols	Optimize for minimum size	Stripped symbols and min size	Stripped symbols, min size, and disabled asserts and memory pools	Percentage Savings
zlib	66805	60676	61844	55472		17%
libxml	802749	403180	781873	398252		50%
glib	1128203	516380	1104001	501744	486492	57%
gobject	720650	269488	702896	259540		64%
gmodule	61608	11400	59892	11292		82%
gthread	68354	16116	68142	16048		77%
mad	203192	92944	193950	89912		56%
Average Savings		55%	3%	56%	57%	57%

Table 6: GStreamer libraries with various compacting options

Optimizing for size had a very small 3% overall impact. Given the anticipated drop in performance, optimizing for size is not recommended using the 2.95.2 gcc compiler with the 10/22/99 ARM patches.

The ARM processor on the target device supports the THUMB instruction set. Typical code size reduction when using the THUMB instruction set compared to the ARM instruction set is approximately 40%. A THUMB GNU compiler was not available for this case study, so no actual code footprint reduction measurements were made.

4.7 Conditional Compilation

GStreamer is a feature-rich multi-media streaming library initially targeted for desktop PC environments. Thus, there are quite a few features in GStreamer that are more appropriate to larger, desktop-style applications. These desktop features have little or no value in an embedded environment. A modification to the build process was needed to allow building GStreamer with all the desktop features and also building with features appropriate for embedded devices. Like the Linux kernel, GStreamer uses conditional compilation to control feature inclusion. The source code was changed to wrap the functionality appropriate only to desktop environments with conditional compile instructions. The conditional compile directives added to the GStreamer source code are listed in the table below.

Conditional Compile Directive	Desktop Feature	Description
disable-loadsave	disable pipeline XML load/save	The load/save feature is only useful for systems that save the structure of a GStreamer pipeline to an XML file and load it back later. An embedded device will almost never need this feature.
disable-typefind	disable typefind plugin	The typefind plugin enables a media player to determine the type of an unknown stream. It uses functions provided by the various codecs to accomplish automatic support of unknown streams. An embedded device only requires this capability if the device supports downloading new codecs.
disable-autoplug	disable autoplugger subsystem	The autoplug subsystem is a collection of elements that enable an application to deal with a media stream without any prior knowledge of the type of the stream. It acts as a 'magic plugin' to convert between two existing elements, searching the list of available plugins for the shortest path. This works best if there are a large number of available plugins. Due to memory constraints, embedded devices are likely to support the appropriate plugin codec rather than use a series of codecs to transform the stream into a supported format.
disable-parse	disable command-line parser	The command-line parser is the core of the gstreamer-launch tool, which enables fast prototyping of many types of pipelines. It may be useful in some embedded situations, because it allows a very concise description of the pipeline to be stored for later use. The GStreamer command line parser is better than XML for simple pipelines.
disable-trace	disable tracing subsystem	The trace subsystem feature is mainly targeted to developers. After development is complete, the tracing feature doesn't add any user features in embedded devices.
disable-registry	disable plugin registry	The registry allows querying a large number of plugins without preloading, which is primarily useful for systems with a large collection of codec plugins.

Table 7: GStreamer conditional compile directives

The optional desktop features with the largest memory footprint are load/save and registry, which when both disabled will remove all dependencies on the XML library, which is also rather large (because it handles Unicode and various other capabilities). Detailed values are shown in the table below savings from both compiler optimization and conditional compiler directives.

Conditional Compile Directive	GStreamer	GStreamer with required libraries	Notes
Starting point	264947	3316508	Required libraries include zlib, libxml, glib, gobject, gmodule, gthread, and mad
Stripped symbols	232080	1602264	
Above with Compilation for minimum size	215272	1547532	
Above with disable-loadsave disable-registry	193836	1072372	No longer need zlib or libxml
Above with disable-typefind disable-autoplug disable-parse disable-trace	180068	1058604	
Above with disable INFO disable DEBUG	160372	1038908	
Above with disable checks disable asserts	106876	895500	glib was not compiled with INFO and DEBUG disabled due to a compiler error. An addition 33% reduction in glib is excepted with INFO and DEBUG disabled
Total percentage savings	60%	73%	

Table 8: GStreamer conditional compile memory reduction

The effort to minimize the GStreamer specific code created a 60% memory footprint savings. However, GStreamer is small (12%) compared to the set of libraries used with GStreamer. Removing the need for zlib and libxml created savings much larger (443 Kbytes) than the GStreamer internal memory footprint changes (85 Kbytes). Effort spent tuning GStreamer without looking at the overall system would be effort poorly utilized.

4.8 Code Changes

In analyzing the generated code, several unexpectedly large code sequences were found. Many of these were trimmed down, and in some cases the change provided a significant performance boost as well.

4.8.1 Object Type Checking Macros

The GStreamer macros of form `GST_IS_<type>(object)` are designed to compare the first four bytes of the object with the appropriate type. The type itself is gotten from `GST_<type>_TYPE`, which typically calls a function that keeps track of a local static variable. The variable is initialized to zero, and gets set the first time the function is called by way of registering the type.

The `GST_IS_<type>` macro is built from `G_TYPE_CHECK_INSTANCE_TYPE`, which originally called the `g_type_instance_is_a()` function. A GStreamer team patch cleaned this up so the

type is directly compared before resorting to the slow function call, but this isn't enough in some cases. Any time the object type does not match the type being checked, the `g_type_instance_is_a()` function is called. An example is checking an instance of a child class to see if it's a member of the parent class.

The resulting check code looks roughly like:

```
if (object->type == type) return TRUE;
else return g_type_instance_is_a(type,object);
```

This compiles into roughly 15 ARM instructions, which is quite large considering these checks are done rather often.

The solution was two-fold:

1. Since `gst_init()` must be called before GStreamer is used, the functions that returns the various object types can be eliminated, moving the type registration into and making reduce to a static global.

2. Implementation of, which does nothing but directly compare the object type, for cases where we only care if it is exactly the type being checked. This reduces the check to:

   ```
   return (object->type == type);
   ```

   which generates inline code and compiles to only a couple of instructions.

4.9 TaskBridge DSP Support

The target device contains both an ARM general purpose processor (GPP) and a TI C55 digital signal processor (DSP). A large amount of media stream processing can and should be offloaded to the DSP. Because of GStreamer's modular design, high-level applications are not aware of which hardware is processing the streaming media.

GStreamer creates a pipeline of the various filters used in processing the media stream. An example of a simple MP3 stream pipeline is a stream reader, MP3 to PCM codec, followed by the audio driver. Pipelines are a common concept within Unix – standard out (`stdout`) from the previous process can be piped to standard in (`stdin`) on the next process. Using Unix style shell pipe command, the MP3 stream pipeline can be represented as shown below with the '|' being the pipe command:

```
$ cat song.mp3 | mp3_to_pcm_codec > /dev/audio
```

The stream reader may be reading the MP3 data from the Linux file system (like shown above) or from the network, both activities best handled by the GPP. The MP3 to PCM codec is ideal for DSP handling. Typically, the audio driver is also controlled by the DSP, so piping the PCM data generated by the DSP codec directly to the audio driver is the most efficient. If the audio driver is under GPP control, then the PCM data must be passed back from the GPP. Even in this simple example, the power of using a pipeline concept in GStreamer is evident.

GStreamer has a standard plugin called 'pipefilter' that is designed to work like the Unix pipe command. pipefilter forks a copy of a the Linux filter program (e.g. the MP3 codec) and attaches to the `stdin` and `stdout` file descriptors. The file descriptors are then attached internally to the sink and src pads of the GStreamer pipefilter element. This allows us to

`fork()` a copy of a program that the kernel will run like any forked process, and tie `stdin` and `stdout` to the GStreamer pipeline. The pipefilter gets us one step closer to doing the main streaming decode on the DSP.

The second step is provided by DSPLinux. TaskBridge is a DSPLinux technology that allows a process running on the GPP to create a process that runs on the DSP, but looks like a standard Linux process to the rest of the programs running on the GPP. We can build a DSP MP3 codec program that takes MP3 data on `stdin` and writes decoded audio to `stdout`. The DSP MP3 codec resides on the Linux file system as an executable file. When GStreamer pipefilter is used as part of the pipeline, it forks the DSP MP3 codec program as described above. TaskBridge identifies the executable as DSP code and handles the necessary setup to get the MP3 codec loaded into DSP memory and running. TaskBridge also makes `stdin` and `stdout` available to the DSP-based MP3 codec.

The current implementation of TaskBridge only handles one DSP-based process. For simple pipelines, this should be sufficient. However, if additional filters are used in a streaming playback — like a parametric equalization filter — being able to fork multiple DSP-based processes would be useful. Even in the simple case of a `cat song.mp3 | mp3_to_pcm_codec > /dev/audio` pipeline, both the MP3 decoder and the audio driver may be on the DSP. TaskBridge must be optimized so the `stdout` data of the DSP MP3 decoder won't require processing by the GPP in order to get to the DSP-based audio driver. These are future optimizations that are not part of the case study demo.

Another possible solution is to have a small version of the GStreamer operational core run on the DSP to enable full GStreamer DSP plugins. The arbitrary containment features of GStreamer could be used to build a "DSP bin" to logically encapsulate elements that run on the DSP and handle data transfer to and from the DSP. This approach works with TaskBridge, which only supports forking a single DSP process. For this case study, TaskBridge was used with GStreamer's pipefilter. Again, this optimization is not part of the case study demo.

5 Linux Kernel Improvements

Memory footprint improvements to the Linux kernel come in several forms. Since Linux is primarily targeted to desktop and server platforms, there are many straightforward approaches that can be followed to reduce the kernel footprint. Several of the methods are explained below, along with specific applications of those methods and measurements of the improvements made. The summary section describes other ways to apply the methods to see even bigger memory footprint savings.

Before diving into the memory improvement methodologies, it is worth describing an actively staffed effort to make Linux work in environments with simple processors – namely the μCLinux (http://www.uclinux.org/) effort and the associated μClibc library. The focus of μCLinux is on processors without memory management units. For this case study, the ARM9 has an MMU and the services the MMU provide make it an important part of processor. However, other memory footprint related changes made to Linux and glibc to derive μCLinux and μClibc could be useful in reducing the Linux footprint. For this case study, we did not refer to μCLinux improvements and instead use "first principles" to identify footprint reduction opportunities.

Referring to the file size of an executable or a library is a poor indicator of what memory resources will be used. A more accurate way is to examine object files and look at the text, data, and bss segment sizes. The text segment contains the code image and the values of constant variables. The data segment contains the initialized data values. The bss segment indicates the size required for the uninitialized data section. The `nm`, `objdump`, and `size` utilities can be used to examine object files created by the compiler or linker.

5.1 Starting Point

The `size` utility outputs the total size of the various segments. When used on the kernel executable, the total sizes of the various segments can be determined. Unless otherwise stated, all memory footprint values are measured on the target device. For this case study, the starting point values are:

```
$ arm-linux-size vmlinux

   text        data         bss         dec         hex     filename
1214980       64276      151736     1430992      15d5d0     vmlinux
```

Listing 1: Kernel starting size for `xconfig`

The `dec` and `hex` columns are the sum of the text, data, and bss segment values in decimal and hexadecimal representation respectively. The value in the `dec` column is used when looking at the total memory requirements of a component or the kernel.

The various improvements were made over several weeks, during which time the code base was improved. For this reason, the starting point values do not match exactly. Since the objective of the case study is to understand the memory reduction improvement, the difference between the improved component/kernel compared to the starting point is the critical information. Using different starting points is unfortunate, but doesn't invalidate the results.

5.2 xconfig – Feature Inclusion Configuration Tool

The Linux kernel is modularized in several different ways. All processor dependencies are isolated so the kernel can be built for use on one of several different processors. For this case study, the kernel is built using the i386 processor code or the ARM9 processor code, as appropriate. Major kernel features are supported in a manner that allows the feature inclusion or exclusion based upon selections made by the user via the xconfig configuration tool. Device driver configuration options include statically linking with the kernel, excluding the driver, or making the driver into a demand loaded module. The *Case Study Starting Configuration* section (page 54) describes the initial configuration used.

The `xconfig` configuration tool was run several times, each time we removed a feature. The kernel was rebuilt after each run to see the changes in kernel size (using the `size` utility). Here are the results:

Change	text	data	bss	dec	hex
Starting point	1214980	64276	151736	1430992	15d5d0
RARP/BOOTP removed	1198884	64136	151704	1414724	159644
CDROM removed	1177932	52020	151704	1381656	151518
NFS support removed	1013348	46724	145992	1206064	126730
Verbose kernel/user messages turned off	1012244	46724	145992	1204960	1262e0
Percentage savings	17%	27%	4%	16%	16%

Table 9: Memory reduction via `xconfig`

Carefully understanding the operating system features needed in the embedded device and excluding all other features using `xconfig` is the simplest method to reduce the Linux kernel memory footprint.

5.3 Static Buffer and Array Size Analysis

Statically allocated RAM is in the data and bss segments. The data segment contains the initial values for the variables and the bss segment indicates the size needed for the uninitialized variables. We can examine the list of data and bss segments with a size greater than or equal to 0x1000 (4096) bytes with the following command:

```
$ arm-linux-nm --size-sort vmlinux | grep -v 00000 | grep " [bBdD] "
00001000 B con_buf
00001000 B pidhash
00001000 b pty_state
00001000 b swap_buffer
00001578 B kstat
00001648 D contig_page_data
00001fe0 b ro_bits
00002000 D init_task_union
00004000 b log_buf
000043f0 B fb_display
00008780 B blk_dev
```

Listing 2: Largest statically allocated variables

If the second column contains a 'b' or 'B', it is a bss segment. Likewise, if the second column contains a 'd' or 'D', it is a data segment. A lower case segment indicator means the segment is local and uppercase means the segment is global.

The kernel code was examined to find where large static arrays were allocated. Since the capabilities of an embedded device are typically much more limited than a desktop or a server, most of these arrays can be made smaller without impacting operation.

Purpose	Initial Value	Modified Value	File
Number of supported disks DK_MAX_MAJOR DK_MAX_DISK	16	1	/include/linux/kernel_stat.h
Maximum block device driver number MAX_BLKDEV	255	31	/include/linux/major.h
kmsg message log size LOG_BUF_LEN	16384	4096	kernel/printk.c
Maximum number of tty consoles MAX_NR_CONSOLES	63	5	/include/linux/tty.h

Table 10: Linux kernel code changes for static variable / array memory reduction

Again, the kernel was rebuilt after each source code modification to see the changes in kernel size (using the `size` utility). Here are the results:

Change	text	data	bss	dec	hex
Starting point	1012244	46724	145992	1204960	1262e0
Number of supported disks: 1	1012180	46724	140888	1199792	124eb0
Maximum block device driver number: 31	1012180	46724	95192	1154096	119c30
kmsg message log size: 4K	1012180	46724	82904	1141808	116c30
Maximum number of tty consoles: 5	1012180	46724	65016	1123920	112650
Percentage savings	0%	0%	55%	7%	7%

Table 11: Memory reduction via static variable / array related code changes

The `nm` utility was run again and the following table contains the comparison with the first run of `nm` listed above for the items that were changed.

Change	Segment Name	Initial Value	Value After Change
Number of supported disks: 1	kstat	00001578	0000018c
Maximum block device driver number: 31	ro_bits blk_dev	00001fe0 00008780	000003e0 00001080
kmsg message log size: 4K	log_buf	00004000	00001000
Maximum number of tty consoles: 5	fb_display	000043f0	00000570

Table 12: bss reduction via static variable / array related code changes

Identifying large kernel static RAM allocations and examining the source code to see if those allocations can safely be reduced is another method for reducing the kernel memory footprint.

In summary, `xconfig` was used to change which features where included in the kernel build, and the following 4 source files were modified:

```
include/linux/kernel_stat.h
include/linux/major.h
include/linux/tty.h
kernel/printk.c
```

Together these changes resulted in the reduced footprint listed in the following table:

	text	data	bss	dec	hex
Starting point	1214980	64276	151736	1430992	15d5d0
After changes	1012180	46724	65016	1123920	112650
Percentage savings	17%	27%	57%	21%	21%

Table 13: Memory reduction via `xconfig` and static variable /array code changes

5.4 Dynamic Memory Usage Analysis

5.4.1 Overview

Kernel routines dynamically allocate memory from the page allocator, slab allocator, or the kernel memory allocator. Analyzing which routines calls the various allocators and identifying those routines consuming large amounts of dynamic memory is a standard method for reducing dynamic memory usage.

5.4.2 Page Allocator

The page allocator is the lowest level memory allocator in Linux. Generally, the kernel does not use the page allocator directly (the file system subsystem is an exception) since the minimum size allocation is one page, or 4096 bytes. Instead the slab allocator is used to more efficiently manage memory. Reducing the file system's dynamic memory allocations via the page allocator was not investigated beyond what is described in the *File System* section (page 29).

5.4.3 Slab Allocator

The slab allocator manages pools of memory for specific purposes. When allocated memory is freed, the slab allocator keeps the memory associated with its pool. Only when the underlying page allocator runs low on memory does the slab allocator release free memory from the various pools. The advantage of this approach is the reduction in time for reallocating memory for the same purpose is much faster than performing all the steps necessary for allocating from general memory. This approach is used because the developers of Linux noticed that the various subsystems freed memory only to later allocate memory for the same purpose.

The current state of the slab allocator can be interrogated via the `/proc/slabinfo` pseudo-file, as shown below. The columns are (1) cache name, (2) cache entries in use, (3) cache entries allocated, (4) cache size, (5) active slabs, (6), number of allocated slabs, and (7), the get free page order flag.

```
$cat /proc/slabinfo
slabinfo - version: 1.1
kmem_cache            57     78    100    2    2    1
nfs_read_data          0      0    352    0    0    1
nfs_write_data         0      0    384    0    0    1
nfs_page               0      0     96    0    0    1
nfs_fh                 0      0     96    0    0    1
blkdev_requests      512    520     96   13   13    1
dnotify cache          0      0     20    0    0    1
file lock cache        0     39    100    0    1    1
fasync cache           0      0     16    0    0    1
uid_cache              0      0     32    0    0    1
tcp_tw_bucket          0      0     96    0    0    1
tcp_bind_bucket        3    113     32    1    1    1
tcp_open_request       0      0     64    0    0    1
inet_peer_cache        0      0     64    0    0    1
ip_fib_hash            5    113     32    1    1    1
ip_dst_cache           0      0    160    0    0    1
arp_cache              0      0    128    0    0    1
skbuff_head_cache      1     24    160    1    1    1
sock                   8     10    800    2    2    1
inode_cache          163    170    384   17   17    1
bdev_cache             4     59     64    1    1    1
sigqueue               0     29    132    0    1    1
kiobuf                 0      0    128    0    0    1
dentry_cache         226    240    128    8    8    1
filp                  48     80     96    2    2    1
names_cache            0      2   4096    0    2    1
buffer_head         8324   8360     96  209  209    1
mm_struct              6     24    160    1    1    1
vm_area_struct       126    177     64    3    3    1
fs_cache               5     59     64    1    1    1
files_cache            5      9    416    1    1    1
signal_act             6      9   1312    2    3    1
```

Listing 3: Slab allocator results

To make the listing more manageable, I deleted the slabs supporting the kernel memory allocator, which is discussed below. The largest four caches are all associated with the file system and block drivers. The four caches are dentry_cache (28928), blkdev_requests (49152), inode_cache (62592), and buffer_head (799104). Adjusting the file system parameters, as described in the *File System* section (page 29), reduces the amount of slab memory allocated.

5.4.4 Kernel Memory Allocator

For this case study, the kernel memory allocator (KMA) was examined in detail. The kernel memory allocator was chosen for analysis because there didn't appear an easy way to identify which kernel subsystem was using KMA. When general kernel memory is allocated and freed, the KMA `kalloc()`and `kfree()` functions are used. These functions operate on a series of geometrically distributed slab sizes from 32 bytes to 131072 bytes. If a memory region larger than 131072 is required, the KMA cannot be used.

To identify any opportunities for dynamic memory usage reduction, the kernel source code was modified. Each subsystem was assigned a unique identifier. The KMA functions were modified to accept the kernel subsystem identifier and track current outstanding allocations and maximum allocations. Also, the number of calls to `kmalloc()` and `kfree()` was tracked. All calls to the KMA where modified to include the subsystem identifier. Finally, a new proc pseudo file system entry was created, `/proc/kmainfo`, to allow access to the KMA usage statistics. The results are shown below (run on the x86 development workstation, not the ARM9 target device).

```
$ cat /proc/kmainfo
kernel memory allocator stats - version: 1.0
subsystem  kmalloc()s  kfree()s   current    maximum    directory
total      33040       32701
kernel     17          3          672        672        arch/i386/kernel
pcmcia     17          3          3008       3264       drivers/pcmcia
usb        34          21         5536       5696       drivers/usb
ide        56          0          5152       5152       drivers/ide
pci        32          0          12320      12320      drivers/pci
sound      3           0          96         96         drivers/sound
char       119         62         46880      46880      drivers/char
net        1           0          1024       1024       drivers/net
ipc        184         173        1344       2080       ipc
core       12416       12414      256        58880      net/core
unix       49          33         960        960        net/unix
mm         15          4          352        384        mm
kernel     137         115        864        896        kernel
fs         19954       19873      22528      28992      fs
autofs4    2           0          96         96         fs/autofs4
devpts     2           0          1056       1056       fs/devpts
ext2       2           0          64         64         fs/ext2
```

Listing 4: Kernel memory allocator results

As can be seen above, the subsystem using the largest amount of KMA is the char driver at 46880 bytes. Since the maximum used by any one kernel subsystem was only around 50K, we did not investigate KMA RAM usage reduction further. The most interesting discovery is the number of `kalloc()` and `kfree()` calls made by the network core (`net core`) and file system (`fs`) kernel subsystems. The numbers above were captured right after boot up, thus the calls made by `net core` and `fs` happened during the boot up process. There may be a boot up performance improvement possible by examining why `net core` and `fs` repeatedly allocated and freed memory. For a consumer embedded device, boot up time will likely be critical.

5.4.5 Swap Daemon Improvements

The starting point was an ARM9 Linux kernel with drivers for the target hardware and other no optimization. Note that these values differ a bit from the other starting point values due to improvements make to the target hardware device drivers that were being developed in parallel with the case study.

```
$ arm-linux-size vmlinux
   text    data     bss     dec     hex filename
1221884   64584  151160 1437628  15efbc vmlinux
```

Listing 5: Dynamic memory allocation starting point

The free utility gives a summary (in kilobytes) of system memory usage:

```
$ free
             total         used         free       shared      buffers
  Mem:        30808         8668        22140            0         6144
 Swap:            0            0            0
Total:        30808         8668        22140
```

Listing 6: Available memory starting point

As can be seen in the previous section, the Linux kernel developers have been fairly careful when allocating static variables, especially statically allocated arrays. Most of the state information managed by the kernel is stored in memory that is dynamically allocated. To get an understanding of how the kernel uses dynamic memory, we have to interact with a running system. Many kernel data structures can be interrogated via the proc file system. The meminfo entry in the proc file system displays the current snapshot of various kernel memory variables.

```
$ cat /proc/meminfo
         total:    used:    free:  shared: buffers:  cached:
Mem:  31547392  8962048 22585344        0  6291456  1089536
Swap:        0        0        0
MemTotal:        30808 kB
MemFree:         22056 kB
MemShared:           0 kB
Buffers:          6144 kB
Cached:           1064 kB
Active:           7208 kB
Inact_dirty:         0 kB
Inact_clean:         0 kB
Inact_target:       28 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:        30808 kB
LowFree:         22056 kB
SwapTotal:           0 kB
SwapFree:            0 kB
```

Listing 7: Dynamic memory information results

Recall that the `cat /proc/meminfo` command listed above was executed by Linux running on an ARM9. The first interesting item to note is the target hardware has 32Mbytes of installed memory. The second item to note is all swap related variables are all zero. This is because the target hardware doesn't have a disk. However, the `kswapd` kernel swap daemon is still running, and using resources. We can disable the kernel swap daemon by modifying the `kernel/vmscan.c` file. A simple way to disable the kernel swap daemon is to comment out the line in `kswapd_init()` that starts the kernel swap daemon, and turn `kswapd()` and `wakeup_kswapd()` into empty functions. Obvious, much more code can be removed, but at this point we are focusing on reducing dynamic memory usage.

The other daemon started in `kswapd_init()` is the kernel reclaim daemon, which frees up pages that are no longer needed. The reclaim daemon is still necessary for demand paging (described in the *Target Device File System Image* section, page 33).

After removing the kernel swap daemon, the `free` command was run again to see the memory savings.

```
$ free
             total         used         free       shared      buffers
  Mem:        30812         8140        22672            0         6144
 Swap:            0            0            0
Total:        30812         8140        22672
```

Listing 8: Memory available after disabling swap daemon

In summary, a significant user of dynamic memory is the kernel swap daemon, which is not used in embedded devices without a disk, as shown in the table below.

Change	Used Memory (Kbytes)
Starting Point	8668
Kernel swap daemon disabled	8140
Savings	528

Table 14: Summary of memory reduction via dynamic memory related code changes

5.5 File System

At first glance, the entire kernel file subsystem could be excluded from the build when targeting embedded devices. A much simpler flash block-read / block-write approach might be sufficient for keeping the few permanent variables used by some embedded devices. Storage devices holding 1024 bytes or smaller probably could implement this approach efficiently.

However, when flash devices get in the 4 Mbytes and larger range, using a traditional file system approach for managing the storage space provides a gain in flexibility that should be utilized. Recall that there is a limit to the number of times a block of flash memory can be written before it is worn out. The Journaling Flash File System (JFFS) contains the algorithms needed to distribute the usage of the flash memory blocks evenly to eliminate problems with a single block being heavily utilized, and thus prematurely worn out. By using JFFS, the developers can concentrate on end user features, not infrastructure technology like prolonging the life of hardware components.

Since the target hardware has 32 Mbytes flash, using the file system to manage part or all of the flash is useful. Therefore we will assume the file system is included in the kernel and we need to examine how to reduce its memory footprint.

5.5.1 File System RAM Usage Analysis

Many buffers are allocated by the file system to hold copies of the information on (or headed to) the mass storage device. Also, the kernel is able to detect when files are being accessed sequentially and the kernel performs a read-ahead so when the application requires the data, the operating system has already pre-fetched it from the mass storage device. In addition, buffers hold copies of the directory structure (`dentry`) and the internal data structure (`inode`) keeps track of where on the mass storage device files are stored. Using all these buffers reduces the number of times information is read from the mass storage device, thus improving performance. For a desktop or server environment, this performance improvement can be substantial.

However, in an embedded device, reading and writing files is not done as frequently. Also, if the embedded device uses flash, reading information is typically faster than with a mechanical disk type storage device. Therefore, reducing the size of buffer caches used appears to be a simple change to reduce the RAM footprint. To test this theory, the various cache sizes were measured as the amount of installed RAM was changed. The values were captured from the messages displayed on the console as the kernel booted (also, the `dmesg` command can be used to capture the values right after the first login). The following table summarizes the findings.

Installed RAM Mbytes	Dentry Cache		Buffer Cache		Page Cache		Inode Cache	
	bytes	entries	bytes	entries	bytes	entries	bytes	entries
16	16384	2048	4096	1024	16384	4096	8192	1024
8	8192	1024	4096	1024	8192	2048	4096	512
4	4096	512	4096	1024	4096	1024	4096	512

Table 15: Cache sizes versus installed RAM memory

The kernel automatically adjusts the buffer sizes based on the installed RAM memory. This means there is not big memory footprint gains by manually changing the cache sizes. As an embedded device under development is nearing completion, small gains could be found by manually tuning the cache sizes.

5.5.2 File System ROM/Flash Footprint Analysis

The kernel file subsystem was examined to see if the ROM/flash footprint could be reduced. The size of the starting configuration (as listed in *Case Study Starting Configuration* section, page 54) is shown below. A second configuration is also shown excluding `ext2fs`, `procfs`, `devfs`, `cdromfs`, `nfs`, `msdos` and, `initrd/ramfs`. The `xconfig` tool was used to change the configuration and then the modified kernel was rebuilt. Since we are not statically linking an application with the kernel, the modified kernel cannot boot, but it is a good reference point to show the minimum possible file system kernel sub-system footprint size.

Change	text	data	bss	dec	hex
Starting point	1221852	64584	151160	1437596	15ef9c
Removed `ext2fs`, `procfs`, `devfs`, `cdromfs`, `nfs`, `msdos` and, `initrd/ramfs`	916644	43936	142280	1102860	10d40c
Percentage savings	25%	32%	6%	23%	23%

Table 16: Case study file system memory reduction

One approach to reducing the kernel file subsystem is to compare the case study size against the size of a different version of the file system. The Linux build process links to gether each subsystem, and a final link of all the subsystems is then performed to generate the `vmlinux` kernel file. The kernel file subsystem is linked into a file called `fs.o`. We can compare various `fs.o` file sizes to get more insight into approaches to reducing the ROM/flash footprint.

Change	text	data	bss	dec	hex
Starting point `fs.o`	376644	23893	8164	408701	63c7d
`fs.o` with `ext2fs`, `procfs`, `devfs`, `cdromfs`, `nfs`, `msdos` and `initrd/ramfs` removed	139246	1740	5060	146046	23a7e
`fs.o` from the uClinux 2.0.38 kernel plus `minix`, `fat`, `vfat`, and `romfs` file systems sizes	151148	12420	18008	181576	2C548
Percentage savings (2.0.38 compared to case study starting point	60%	48%	21% larger	56%	

Table 17: Case study versus uClinux 2.0.38 file system comparison

From the above table, the total size of a usable 2.0.38 uClinux file system (181576) isn't that much larger that a completely stripped down case study file system. One strategy to reducing the file system ROM/flash footprint is to use the file system from an older version of the kernel. This approach is discussed more in the *Summary* section (page 47). Using the file system from the 2.0 version of the kernel could offer good memory savings. However, there will be linking problems that must be carefully analyzed to resolve correctly. The next section discusses the linking problem details.

5.6 Networking

One of the main reasons that there is so much interest in Linux as an embedded operating system is due to its great network support. Other operating systems targeted to embedded applications and home grown embedded operating system have little or no network support.

For this analysis, we started with networking turned off, and then added functionality, as shown in the table below.

Change	text	data	bss	dec	hex
Starting point – networking turned off	742420	42732	139308	924460	e1b2c
Kernel with networking turned on, but no TCP/IP or drivers (just 802 and network scheduling)	795684	50652	219600	1065936	1043d0
Kernel w/networking,TCP/IP, and SMC9194 driver	998012	50980	223124	1272116	136934

Table 18: Size of case study kernel with various networking capabilities enabled

Size information for uClinux 2.4.10 kernel builds with similar configurations were also captured in the following table.

Change	text	data	bss	dec	hex
Starting point – networking off	376488	18980	123492	518960	7eb30
Kernel with networking turned on, but no TCP/IP or drivers (just 802 and network scheduling)	409912	22368	124128	556408	87d78
Kernel w/networking,TCP/IP, and SMC9194 driver	626672	25260	127984	779916	be68c

Table 19: Size of 2.4.10 uClinux kernel with various networking capabilities enabled

Unfortunately, a different .config file was used when building the of 2.4.10 uClinux kernel, so comparing these results with those of the 2.0.38 kernel are not entirely meaningful. However, the size differences are compared in relative terms below.

First we build the uClinux 2.0.38 kernel under the same networking configurations and captured the following results.

Change	text	data	bss	dec	hex
Starting point – networking off	433744	41076	65280	540100	83dc4
Kernel with networking turned on, but no TCP/IP or drivers (just 802 and network scheduling)	445312	43552	66816	555680	87aa0
Kernel w/networking,TCP/IP, and SMC9194 driver	557152	47724	70912	675788	a4fcc

Table 20: Size of uClinux 2.0.38 kernel with various networking capabilities enabled

The results comparing the case study, the 2.4.10 uClinux, and the 2.0.38 kernel are summarized in the table below.

Kernel	Networking turned on Delta size	Networking with TCP/IP and SMC9194 driver - Delta size
Case study	141476	347656
2.4.10 uClinux	37448	260956
2.0.38 uClinux	15580	135688

Table 21: Comparison of various kernels with various networking capabilities enabled

The table lists the delta size of the kernel (column labeled dec) compared to the kernel size with networking disabled. Around a 100 Kbyte saving is possible using the uClinux 2.4 network stack instead of the regular 2.4 network stack used in the case study, with most of the saving being in reduced RAM usage. If the 2.0 networking features are sufficient, 200 Kbyte savings are possible by using an older version of the uClinux network stack.

We replaced the case study kernel file subsystem with the uClinux 2.0.38 code. The code compiled without error. However, during the linking phase, 43 externals were unresolved, as listed in Table 22. Each unresolved external requires investigation and a code changes made, followed by through testing. Using an older kernel subsystem with the case study kernel demonstrates that interfaces and shared variables internal to the kernel are changed as the kernel developers improve the code. However, the external kernel interfaces, those used by applications, are very stable.

```
arp_broken_ops
arp_tbl
csum_partial_copy
__csum_partial_copy_fromuser
current_set
datagram_select
dev_alloc_skb
devinet_ioctl
dev_kfree_skb
dev_lockct
icmp_reply
in_dev_finish_destroy
inet_addr_type
inetdev_by_index
inetdev_lock
inet_dgram_ops
inet_family_ops
inet_select_addr
intr_count
ip_cmsg_recv
ip_finish_output
ip_mc_dec_group
ip_mc_inc_group
ip_options_undo
ip_route_input
ip_route_output_key
__ip_select_ident
ipv4_config
kerneld_send
kfree_skb
proc_net_inode_operations
proc_register
putname
register_inetaddr_notifier
skb_device_lock
skb_device_locked
skb_device_unlock
sock_wspace
tcp_set_keepalive
unregister_inetaddr_notifier
verify_area
wake_up
wake_up_interruptible
```

Table 22: Unsolved file system variables / function calls

Looking at the changes to the files between 2.0 and 2.4, some 2.4 files can easily be slimmed down. Multicasting support in `igmp.c` could be wrapped with conditional compile directives, and thus easily excluded. But large files such as `tcp_input.c` have a lot of functionality that can't be easily removed.

5.7 Network Tuning

There are networking tuning parameters that can be adjusted (see `linux/net/TUNABLE`). In the `include/linux/netlink.h, MAX_LINKS` is set to 32. We lowered the value to 3 to see what additional memory footprint reduction was possible. No useful memory reduction was measured.

5.8 Target Device File System Image

The awkward phrase “kernel file subsystem” used previously was referring to the kernel code implementation that supports the file system functionality. In this section, we talk about the “target device file system image” when referring to the contents of the file system on the target device. The target device file system contains all executables and data except for the boot loader and the kernel itself.

When examining ROM/flash footprint reduction, the Linux file system plays a unique role that doesn’t match historical approaches to software in embedded devices and doesn’t quite match the typical desktop PC view of a file system either.

Historically, embedded device software was designed to control the device in a specific fixed manner. The “application” software that provided the high level functionality for the device was statically linked with the operating system kernel and the resulting single executable was put in ROM, or more recently in flash. When the device was turned on, the processor’s reset vector

pointed to the executable and that caused the single executable to run. The software in most existing embedded devices has no concept of separate executables residing in a file system.

On the other hand, the notion of a file system on a desktop PC is a place to store programs and data. When a user issues a command, either via the a command prompt input or GUI windowing manager invocation, the computer accesses the file system to get the information necessary to carry out the request.

Embedded devices running Linux can use the notion of a file system in a very powerful way to reduce the size of the ROM/flash footprint. This approach works because Linux uses a demand paging scheme via the processor's memory management unit (MMU). Demand paging works by only loading the first page of a program into memory and then transferring control to the program. As the program runs, it makes references outside the page or transfers control to code residing on a different page. In either case, a page fault occurs when the MMU detects the page containing the requested information is not loaded. The kernel handles the exception by loading the appropriate page (and configuring the MMU data structures approximately). The program can then continue running. This means the entire program can run without being completely loaded into RAM.

Code in ROM/flash can either be executed-in-place (XIP) or copied into RAM and then executed. Obviously, copying the code into RAM can be time consuming and uses more RAM. However, with today's flash memory, code executes significantly faster out of RAM compared to flash. A second advantage is the additional power savings by being able to run the code in less time, due to fewer wait states, and thus extending battery life.

Linux brings a third advantage for executing code from RAM. Demand paging means that each page that makes up the file system can be compressed as the flash image is being built. When a page fault occurs, the exception handler locates the compressed page in the flash memory, decompresses into RAM, and then execution can continue as before. The big savings using this approach is eliminating the long wait for an entire program to be decompressed before execution begins and anything stored in the file system can be compressed. In Linux, CRAMFS supports this technology.

A master of the file system to be compressed is traditionally made on the development workstation. CRAMFS uses the zlib compression library. The `mkcramfs` tool compresses the master file system into a file, normally ending with .crm. The boot loader, kernel, and compressed file system are then transferred into the target device's flash memory. The boot loader starts executing on power-up, loads the kernel into RAM (which normally is uncompressed in the process). Linux boots up and mounts the compressed file system. At each page fault, the appropriate page image is decompressed before being loaded into RAM.

The master version of the file system used by DSPLinux for the target hardware was created on the development workstation. The `du` utility was used to see measure the master file system size.

```
$ du –hs fs

   3.8M fs
```

Listing 9: Uncompress file system image size

Running `mkcramfs` and checking the compress file system size can be done as follows:

```
$ mkcramfs fs fs.crm
$ ls -la fs.crm
-rw-r--r--    1 stevej   users     1732608 Oct  2 12:49 fs.crm
```

Listing 10: CRAMFS compress file system image size

`mkcramfs` also provides a detailed output for each file. The output is summarized, with files changed by less than 100 bytes excluded from the list (negative numbers means the file was compressed).

Percentage Change	Size Change (in bytes)	File Name
-42.64%	-89736	busybox
-49.45%	-182220	tinylogin
-64.68%	-5340	hwerror.o
-50.71%	-44624	ld-2.1.3.so
-59.51%	-1928	libBrokenLocale-2.1.3.so
-58.50%	-5368	libSegFault.so
-54.18%	-514388	libc-2.1.3.so
-53.97%	-10792	libcrypt-2.1.3.so
-52.06%	-142496	libdb-2.1.3.so
-48.09%	-26268	libdb1-2.1.3.so
-59.36%	-6200	libdl-2.1.3.so
-49.46%	-80820	libm-2.1.3.so
-58.38%	-50684	libnsl-2.1.3.so
-53.64%	-23368	libnss_compat-2.1.3.so
-64.49%	-15572	libnss_db-2.1.3.so
-52.55%	-5980	libnss_dns-2.1.3.so
-60.93%	-23384	libnss_files-2.1.3.so
-56.29%	-7588	libnss_hesiod-2.1.3.so
-59.17%	-24068	libnss_nis-2.1.3.so
-59.98%	-28132	libnss_nisplus-2.1.3.so
-72.71%	-56804	libpthread-0.8.so
-52.06%	-26556	libresolv-2.1.3.so
-58.23%	-7968	librt-2.1.3.so
-63.21%	-167020	libstdc++-3-libc6.1-2-2.10.0.so
-62.13%	-10872	libthread_db-1.0.so
-58.11%	-4600	libutil-2.1.3.so
-54.97%	-2373	ldd
-50.32%	-4448	ftpcount
-64.03%	-6687	xferstat
-47.66%	-31844	ftp
-48.72%	-86300	pgserver
-53.60%	-18154	libpgui.a
-38.77%	-233	libpgui.la
-38.87%	-234	libpgui.lai
-49.76%	-68432	xinetd
-55.13%	-24176	telnetd
-52.16%	-3964	ckconfig
-51.38%	-4244	ftprestart
-52.84%	-5912	ftpshut
-51.35%	-4952	privatepw
-50.95%	-93216	wu-ftpd
-50.24%	-27136	boa
-55.49%	-3915	boa.conf
-54.56%	-413	test-cgi
-44.12%	-379	mime.types
-53.05%	-226	linuxrc

Table 23: CRAMFS detailed file size reduction

In summary, when execute-in-place performance issues exist, using CRAMFS can provide a significant reduction in the amount of ROM/flash required, as shown in the table below.

Change	Used Memory (Kbytes)
Target device file system image starting Point	3876
`mkcramfs` compress file system image	1692
Percentage savings	56%

Table 24: CRAMFS file system image size reduction

As mentioned previously, CRAMFS uses zlib (http://www.gzip.org/zlib/), a compression / decompression library whose compression algorithm is based on PKZIP. The zlib library is a good general purpose compression algorithm. However, the data in the file system is not general purpose data. The majority of the files contain ARM executables. Improved footprint and decompression performance can be obtained by substituting a compression algorithm tuned for the ARM instruction set. Since the data is only compressed once and decompressed as needed by demand paging, the algorithm can be designed to favor ease of decompression over ease of compression. This approach can give both ROM/flash footprint improvements as well as demand paging decompression performance improvements.

5.9 Subsystem Replacement

The majority of techniques for reducing the memory footprint listed above are straightforward approaches that most development engineers can apply when appropriate. The advantage of the above approaches is they can often be applied to a newer version of the kernel without difficulty.

Another approach is also possible. If additional memory reduction is required, substituting replacement code for major subsystems can be considered.

There are several significant disadvantages to replacing major subsystems. First, it creates a fork in the code. Improvements on the main source code branch are typically difficult to incorporate into the forked code. Second, testing of the branched code is less encompassing since fewer people are using the branched code. Third, supporting the branched code is problematic and typically expensive. Fourth, gaining sufficient marketing momentum to make the branched code a commercially viable activity is difficult.

When possible, it is most effective to develop improvements that not only meet specialized requirements, but better meet the desktop/server needs as well. Such changes would have a better chance of being incorporated into the main code and thus avoid the problems associated with forked code.

Each of the major subsystem is examined for anticipated savings based on using a code replacement strategy.

5.9.1 File System

Linux supports a large number of host file formats by using an intermediary – the Virtual File System (VFS). Linux 2.4.9 ships with support for around 25 host file formats. If the embedded device requires only one host file format, the multi-layered file system can be replaced with something much simpler. Devices that support removable media or need to execute out of RAM will likely need multiple host file formats, and thus will not be able to use this approach.

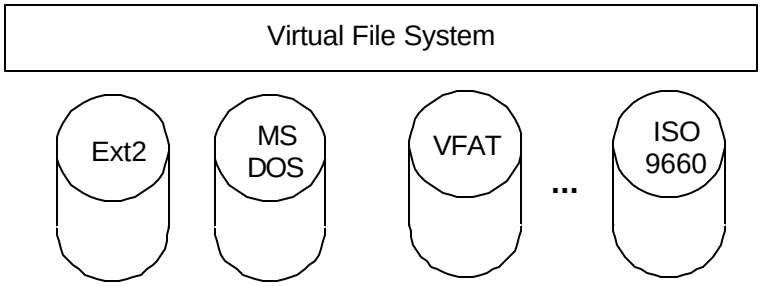

Figure 3: Virtual File System Example

Simplifying the file system to only support a since host file format is estimated to take 3 engineering months of effort. The memory footprint reduction is estimated to be 40 Kbytes RAM and 220 Kbytes flash/ROM. In such a scenario, the possible risk is that applications that depend on the proc file system would not function. Since most application that use the proc file system are utility applications, this limitation should be manageable.

5.9.2 Module Support

Many more I/O ports and peripherals exist than are supported on a single hardware platform. To efficiently use memory, Linux can support device drivers for I/O ports and peripherals as loadable modules. Since the hardware on an embedded device are known in advance and typically do not change, technically the entire module support code can be removed.

However, any device drivers that are statically linked with the kernel will need to have the source code released under the terms of the GPL. There are scenarios where the supplier of a device (whether as a discrete peripheral, as part of a SOC, or as part of licensed hardware IP) would prefer to implement a device driver and make it available without disclosing the source code, and loadable module approach allows such drivers to still be incorporated in a product.

The Linux configuration system already supports building a kernel without module support. The device would run marginally faster (due to the removal of one level of indirection). The memory footprint reduction is estimated to be 37 Kbytes flash/ROM with the RAM requirements essentially unchanged. Compatibility would not be compromised in this scenario.

5.9.3 Memory Manager

Memory management is the largest subsystem within the Linux kernel. The main components of the memory management subsystem, plus part of the glibc library that supports application memory management related functions, are shown below.

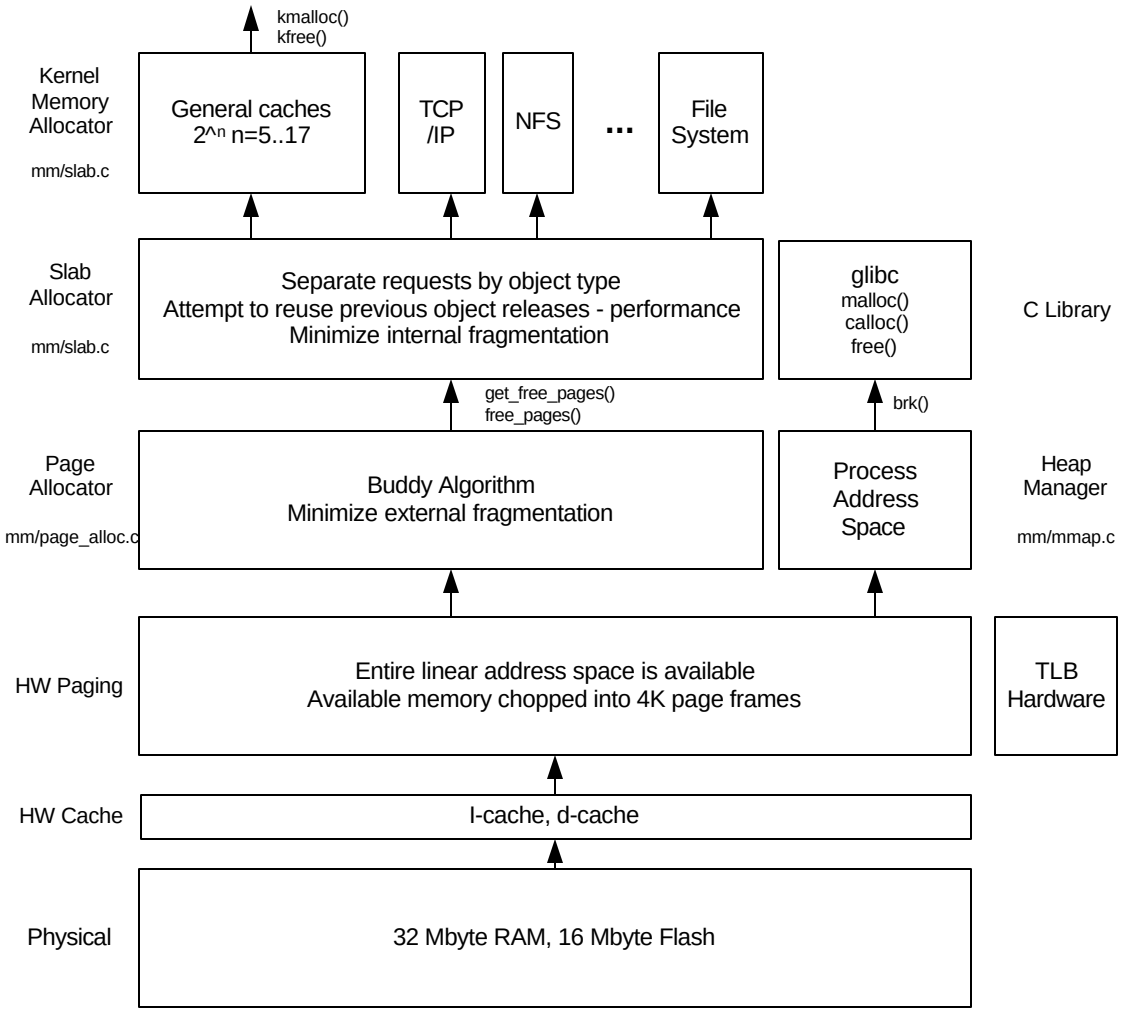

Figure 4: Linux memory management components

The page allocator and slab allocator can be replaced with a simpler memory management scheme if the embedded device is self-contained (i.e. no removable storage, no networking) and the applications are not very sophisticated. However, under these conditions, a simpler operating system may be a better choice than Linux. There are scenarios where a device (such as a lower-end in a family of devices that also run Linux) can benefit from the applications base of other family members of the device. In this case, the following strategy applies.

Replacing the page allocator and slab allocator is estimated to take 8 engineering months of effort. The device would run marginally faster (due to the removal of several layers of memory management). The memory footprint reduction is expected to be 20 Kbytes RAM and 50 Kbytes

flash/ROM. Memory fragmentation issues with the replacement memory manager would need to be characterized.

5.9.4 Networking

Linux's strength is networking. Most new protocols are first deployed on Linux. However, the advanced features supported by Linux increase the memory footprint. Evidence for this was shown in the *Networking* section on page 31, where the memory footprint for the networking subsystem in uClinux 2.0 was compared to 2.4.

Replacing the networking subsystem with an earlier version is estimated to take 4 engineering months of effort. The memory footprint reduction is expected to be 50 Kbytes RAM and 200 Kbytes flash/ROM. Compatibility with the newer networking features, like those included in IPv6 support, would be sacrificed. *(check into this…does kernel 2.4 include IPv6, or IPv4? If IPv4, then what is sacrificed comparing this stack to an earlier one? If IPv6, then the comment can stand as stated)*

5.9.5 Kernel

The interesting features for embedded devices supported by the kernel are module support (described above) and the scheduler. The scheduler is small (6K ROM/flash, 7K statically allocated RAM), so changing the scheduler will not make significant reduction in memory footprint. However, the time-sharing scheduling policy supported by Linux is complex and can lead to poor embedded process-switch characteristics. Several replacement scheduling policies are available that may be better suited to embedded devices with soft real-time requirements. However, since this case study focuses on memory footprint reduction, replacement scheduling policies was not investigated.

5.9.6 Interprocess Communication

Interprocess communication (IPC) is the smallest Linux subsystem. The kernel configuration system already supports building a kernel with most of the IPC code removed. Full Linux IPC (with System V IPC support) is around 28K ROM/flash. Excluding System V IPC support reduces ROM/flash requirements to less than 1K.

5.9.7 The glibc Library

The biggest opportunity to reduce the memory footprint requirements by making implementation changes lies with the glibc library. The case study glibc library requires 870K ROM/flash and 138K RAM.

Various proposals have been made for run-time library alternatives to glibc for use in embedded devices based on Linux. Primarily, these are variants of POSIX 1004.1, however, they are identified in specification form rather than an actual implementation. According to the Embedded Linux Consortium, Red Hat had made a proposal in mid 2001 to provide such an implementation based on their EL/IX "level 3" spec, and to make its implementation available for all developers under an open source licensing model. However, they included a requirement to retain copyright and ownership on the library. As such, the proposal was rejected by the ELC and the offer was withdrawn. Its implementation was suspected to be based upon newlib with enhancements to support pthreads and to be made thread-safe.

Creating an embedded version of glibc is a very large task. For such an engineering effort to be successful, it would likely have to be done by a company experienced in generating open source code. The effort would cost at least $750K and take 6-9 months for an open source embedded version compatible with glibc to be developed and tested. The embedded version would require around 250K ROM/flash (saving around 620K) and 75K RAM (saving 63K). Likewise, for such an effort to be commercially successful, it would also have to have a self-sustaining business model. We have suggestions for creating such a successful model that includes best practices on the commercial side as well as engineering side.

6 glibc Improvements

The traditional Linux C library, called glibc (http://www.gnu.org/software/libc/libc.html), is developed by the GNU community. The glibc library connects applications with the kernel system calls and also supports many commonly used C-language functions. For the case study, glibc is a shared library – only one copy of glibc is needed to service all the applications. However, glibc is also very large, as shown in the table below (based on the `dec` file from the `size` command).

Version	libc	Linux kernel (compressed)
x86 development workstation running Linux 2.4.2 with libc 2.2.2	1247697	793965
Case study starting point	923453	488360

Table 25: libc size comparison

Many of the techniques show in the case study can be applied to glibc to reduce its RAM and ROM/flash memory footprint. In addition to those approaches, several approaches unique to libraries are possible, as described in the following sections.

6.1 Alternate C Libraries

The glibc library, like many other components used with Linux, is targeted to and most utilitized on the desktop and server environments. As can be seen in Table 25, glibc is a large library. Several other C libraries are available, including newlib, uClibc, and diet libc.

6.1.1 newlib C Library

The newlib library is a C library intended for embedded system use and is typically distributed by Cygnus/Red Hat as the target library with its commercial GNU tool chains. It is a conglomeration of several library parts that make them easily usable on embedded products. However, newlib is not thread-safe, and as such, will not support various glibc features such as reentrancy. Additionally, it does not support pthreads, which is a pre-requisite when attempting to use the "gdbserver" program (a useful utility for application-level debugging). It is delivered in source and binary form from various commercial sources, and available in source form on the web. the newlib library has been compiled and stabilized for a wide array of target processors, and will usually work on any architecture with the addition of a few low-level routines.

Starting with the newlib sources, we attempted to build an ARM9 library. The newlib library uses a configure script to determine system dependent variable values. Given that newlib is designed for use in embedded systems, it is odd that the build environment is not cross development friendly. We were unable to build newlib for an ARM9 processor in the time available.

6.1.2 uClibc C Library

uClibc is a C library for embedded systems and is typically used with uClinux. It is an active development project and we use uClibc in our ARM7 solutions. uClibc RAM and ROM/flash memory size information is provided in various places throughout this case study.

6.1.3 Diet libc C Library

The diet libc is a C library optimized for small size. Diet libc can be used to create small statically linked binaries for Linux on a variety of processors including ARM9. Since the case study doesn't statically link applications to the libraries (it uses shared libraries), memory footprint analysis on diet libc was not performed. However, it may be a very viable option where static linking is employed. Also, Diet libc license is GPL – therefore, statically linking to Diet libc requires the resulting code to also have a GPL license.

6.2 Removing Unused Library Functions

Because of the nature of shared libraries, the functions in the library are included even when they may not be used by any of the applications. Some embedded devices ship with a fixed set of applications; new applications cannot be added. In these fixed capability devices, unused library functions increase the memory footprint without adding any value. Various library compression tools are available to strip unused functions out of shared libraries to reduce their memory footprint.

One such open source tool is the Library Optimizer (http://sourceforge.net/projects/libraryopt/). The Library Optimizer tool rebuilds shared libraries to contain only the object files needed to provide symbols required by executables and shared libraries in a given directory tree. It can be used to reduce file system sizes for embedded systems. Library Optimizer is used as a replacement process for the last step in library creation where the parts that comprise the library are actually linked together. Library Optimizer builds a list of dependencies so only those functions with a dependency on the applications to be used are included in the build process.

The Library Optimizer was used to optimized libc version 2.1.3 from the GNU glibc suite of libraries for the case study set of applications. During development, we use a version of the libc library that contains symbols used for debugging. A smaller version of the libc library can be created using the `strip` utility which simply removes sections containing debugging symbol information. The stripped version still contains the full set of functions and data structures. The first two entries in Table 26 show the full libc library with and without symbols.

To determine the maximum libc optimization possible using tools like Library Optimizer, the libc library was optimized to contain only those functions required to make the following program runnable.

```
void main( void ) { }
```

Figure 5: Program simplest.c used for maximum libc library optimization

Notice the program, named `simplest.c`, appears to be completely empty. However, we can interrogate the compiled version, `a.out`, to determine the list of unresolved externals:

```
$ arm-linux-objdump -T a.out
a.out:     file format elf32-littlearm
DYNAMIC SYMBOL TABLE:
02000230      DF *UND*  00000220  GLIBC_2.0   abort
02000240      DF *UND*  00000198  GLIBC_2.0   __libc_start_main
02000380 g    DO .rodata      00000004  Base        _IO_stdin_used
```

Figure 6: List of unresolved externals in compiled version of simplest.c

Even though `simplest.c` appears to be empty, the `abort()` and `__libc_start_main()` functions are called and the `__iostdio_used` data structure is accessed. When using Linux shared libraries, an ELF shared library program interpreter library is required, called `ld-linux.so.2` for our example. Running Library Optimizer on the libc library on a file system that contains the complied version of `simplest.c` and `ld-linux.so.2` created an optimized libc library. The optimized library required 173 separate object files. Generally, each file in the glibc source contains one function. Therefore, to completely resolve all externals required by `simplest.c, ld-linux.so.2` and the functions / data structures used in the libc library requires around 173 libc functions. These functions include most of the common file system functions (open, read, write, …), string functions (strcpy, strlen, …), memory copy functions (memcpy, memset, …). These functions are required because every program has a standard in, standard out, and standard error file handles, plus associated functions called by the related file system functions. The size of the minimum libc is also shown in Table 26.

Library Optimizer was again run on the case study target file system image, with the results shown below.

Change	Code (text)	Initialized data (data)	Uninitialized data (bss)	Total (dec)	File Size
libc with symbols	890253	19028	14168	923449	3863993
libc with symbols stripped	890253	19028	14168	923449	949480
Minimal libc	219037	7660	3328	230025	234564
libc to support a GUI, MP3 player and a web browser	537911	14032	9220	561163	571576
Percentage savings for GUI, MP3 player, web browser build	40%	26%	35%	39%	

Table 26: Library Optimizer results

For the past 20 years, microprocessor code in embedded devices has largely been fixed at the factory with no ability to enhance the device's capabilities by adding or replacing software. Limited, expensive and poor connectivity made adding software difficult. In addition, the software infrastructure in a device to support software downloads is complicated.

With cheap memory and a powerful operating system like Linux, the rules have been changed. Linux supports the concepts needed for software download and execution. The software for embedded devices can be viewed as multiple applications able to dynamic link to the underlying system (the C library and any other libraries).

If an embedded device supports software download capabilities, what happens if the shared libraries were optimized by removing functions not used by the base set of applications? There are several possibilities listed in the following table.

Case	Implication
Necessary functions are available	A new application can be run properly if the application limits the use of system calls to the reduced set supported in the embedded device. Using this approach requires careful understanding of which functions were discarded and building and testing the new application in an environment containing libraries with a similar function set. Managing new application development to a non-standard set of functions requires effort in areas that are not valued by the customer.
Application builds missing functions into the application	The application can include the missing functions. This statically links the application to the missing functions. If another new application needs one of the missing functions, it will have to also statically link in the function even if another application has the same statically linked function. Flash memory is not utilized optimally when several new applications support the same statically linked functions. However, the inefficiency is typically small if the majority of the embedded devices are not enhanced or upgraded (which is a common scenario). This approach creates difficulties with managing and integrating the missing functions.
Replace the functionally reduced C library	A simple approach is include the full C library as part of the new application download process. Using this approach appears to negate the advantage in not shipping the full C library in the first place. However, if the new application requires additional storage to be added to the device (like compact flash, SmartMedia, etc), then the base device has a small memory footprint fitting into the base memory with additional applications requiring additional memory be installed.

Table 27: New applications working with a functionally reduced library

The focus has been on removing unused C library functions. However, this approach may be even more valuable removing functions from more specialized libraries. GStreamer uses the GObject library. Library Optimizer can be used to remove GObject functions not required by GStreamer. It is unlikely other applications will also use GObject, so the missing functions may not be an issue.

At first it appears that removing unused library functions required for just one application could be accomplished by statically linking the application to the library. Technically this is true. However, the code license, like LGPL, may have different requirements based on static versus dynamic linking.

7 Audio Playback Performance Analysis

The load on the ARM9 was measured with the ARM9 performing the MP3 decode and the results were compared with the ARM9 load when the DSP was performing the MP3 decode. The load was measured using a Lauterbach Trace32 JTAG debugger performance tool with the Linux kernel and the file system loaded into RAM. A different performance model would be captured if the kernel and the file system were loaded into flash memory, but the relative CPU utilization difference related to MP3 decoding would be similar.

The MP3 software chosen to run on the ARM CPU, `mpg123`, was originally written for x86 desktop hardware, requiring a minimum of a 486 running at 120Mhz. When executed on the ARM CPU in the EVM at 60 Mhz, the algorithm couldn't keep up with audio playback. Steps that could be taken to get the ARM CPU based MP3 decoder running at speed include:

- Using an MP3 decoder algorithm tuned for the OMAP hardware, paying careful attention to when data cache misses occur
- Run the ARM CPU in the OMAP 1510 at a higher clock rate (it is designed to run up to 175 Mhz – the EVM speed is 60 Mhz)
- Use the 192Kbyte SRAM internal to the OMAP 1510 instead of the slower off-board SDRAM.

The ARM CPU in the OMAP 1510 is easily capable of decoding MP3 at speed. Since the goal of the case study demo is to show how much easier it is to process real time streams using a DSP, no effort was expended optimizing the ARM based MP3 decoder.

7.1 ARM Idle CPU Utilization

A screen snapshot for Linux running on the OMAP 1510EVM sitting idle is show below. Notice 51.1% of the time the CPU is in the `arch_idle` function and 48.7% of the time the CPU is in the `cpu_idle` function.

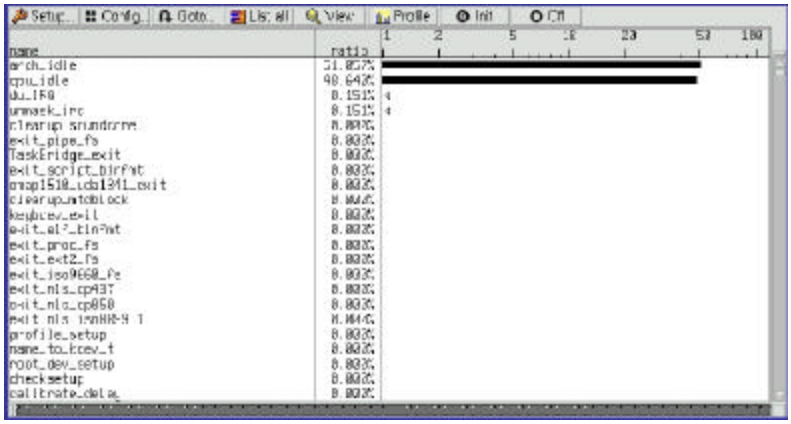

Figure 7: ARM CPU utilization when idle

7.2 ARM MP3 Playback CPU Utilization

Several MP3 decoders were tested. MAD, the MP3 decoder used with GStreamer in the demo software was tested, along with MPG123. At the time these performance tests were conducted, the MPG123 decoder had better performance characteristics, so MPG123 was used. Since the 1510 has a DSP that can be used for MP3 decoding, no effort was spent trying to optimize the ARM9 based MAD or MPG123 decoders.

The MP3 decoder operates in two stages. In the first stage, large collections of constants are generated. These constants are generated using floating point math implemented in software. The CPU utilization for the `mpg123` constant generation stage, based on the ARM9 running at 60 Mhz, is shown in the following table.

Linux Kernel Routine	Percentage CPU Utilization
`mpg123`	13.8
`float64_mul`	13.5
`DoubleCPD0`	12.0
`roundAndPackFloat64`	10.9
`EmulateAll`	8.5
`EmulateCPD0`	7.2
`subFloat64Sigs`	4.3
`addFloat64Sigs`	4.2
`PerformLDF`	4.1
`checkCondition`	2.6
`SetRoundingMode`	2.2
`PerformSTF`	1.8
`normalizeRoundAndPackFloat64`	1.7
`EmulateCPDT`	1.6
`float64_sub`	1.3
`getDestinationSize`	1.3
`float64_add`	1.0

Table 28: CPU utilization for MPG123 constant generation

The `mpg123` program utilized 13.8% of the ARM CPU during the constant generation stage. Essentially the rest of the time the ARM CPU was executing floating point math operations for `mpg123`.

After completing the generation of the required constants, the mpg123 decoder starts converted the MP3 data into PCM data. The MP3 decode CPU utilization, again based on the ARM9 running at 60 Mhz, is shown below.

Linux Kernel Routine	Percentage CPU Utilization
`mpg123`	98.5
`timer_bh`	0.3
`__wake_up`	0.2
`schedule`	0.2
`Update_wall_clock`	0.1

Table 29: CPU utilization for MPG123 MP3 decoder

For 98.5% of the time, the ARM CPU executed functions in the `mpg123` program. The `mpg123` program only used integer math operations during decode, so we don't see any calls to the floating point functions. If an ARM9 optimized version of `mpg123` program was used, we would also see some CPU time spend in the idle routines. The unoptimized version could not meet the required output rate necessary for acceptable quality audio due to pauses caused by the `mpg123` program requiring more procession power than available.

7.3 C55 DSP MP3 Playback CPU Utilization

The same MP3 file was decoded using an Imagine Technologies C55 MP3 decoder on the C55xx DSP core. The Lauterbach performance monitoring tool was again used to monitor the ARM9. No performance monitoring was done on the DSP. The ARM CPU utilization is shown below.

Linux Kernel Routine	Percentage CPU Utilization
`cpu_idle`	50.2
`arch_idle`	45.6
`TaskBridge`	0.3
`do_generic_file_read`	0.1
`do_IRQ`	0.1
`fixup_irq`	0.1
`schedule`	0.1
`omap1510_leds_event`	0.1
`file_read_action`	0.1

Table 30: CPU utilization for Imagine Technologies C55 MP3 DSP decoder

The vast major of the time the CPU is idle when the decoding is done by the DSP. There is some CPU time required by the `TaskBridge` program feeding the DSP algorithm (0.3%), plus some time in the file system retrieving the MP3 file (`do_generic_file_read` at 0.1% and `file_read_action` at 0.1%).

The DSP MP3 decoding load was not directly measured. The Imagine Technology MP3 decoder algorithm specifications are provided below:

Program Memory (k word)	< 16k
Data Memory (k word)	< 7.5k
Processor Loading (MIPS)	< 35

Table 31:Imagine Technology C55 MP3 Decoder Specifications

Although the numbers show the advantage of using a DSP, there is another advantage that is apparent when using the demo code running on the OMAP 1510 EVM. We observe this when we begin playing an MP3 file using the ARM and simultaneously browsing the web. Most prominent is the increase in the playback delay. Repeating this procedure using the DSP to execute the MP3 decode, we notice that the browsing performance is improved with no degradation in audio quality.

8 Summary

In review of the quantitative data of code size reductions illustrated above, there are some notable results that we see, some of which were anticipated, and some of which were a surprise. Additionally, insight drawn during the process of collecting this information leads us to make some conclusions and assertions for the best strategy for system optimizations when assembling software components for the creation of an embedded device.

The greatest reductions, as expected, were in excluding Linux capabilities that are obviously unnecessary in a non-server application. These are effectively "systematic" changes that can managed when basing an embedded design on a commercially derived distribution. The resulting difference in functionality would effectively eliminate attempts by the program to perform certain tasks or allow for certain conditions that are not applicable to an embedded device. Examples of such are as follows:

- Reductions from `xconfig` modifications: 16% nominally from gross removal of non-embedded components.
- Modifications to the users the kernel memory allocator: tightening of `core` and `fs` calls to `kalloc()` and `kfree()` to improve execution time, but may result in minimal improvement in code size.
- Disable kernel swap daemon: reduction by 528K bytes. Big savings in RAM.
- RAM reductions from Static Buffer changes: approx 50%
- Program size reduction from file system changes: approx 25%.

As a result of these reductions, there can be a reasonable reduction in hardware resources (primarily RAM and ROM/flash) allowing a device to reach a new threshold in functionality vs. cost.

Some of the code reduction opportunities that were pursued that did *not* result in significant code reduction were in the following categories:

- Compiling for minimum GStreamer code size: approx 3%

Making such changes may not result in significant size savings, but may still be pursued for the sake of final system optimizing and tuning.

Additionally, there were some "non-systematic" approaches to kernel size reductions that include the risk of fragmentation or imposing modifications that do not also translate into modifications that are welcome by the core user base of the Linux operating system (namely, in servers and IA32 architecture workstations). Some of the approaches considered could result in large code reduction, but at the expense of significant engineering effort and potential quality issues. These options include:

- Replacing the Linux 2.4 kernel file subsystem with the one from Linux 2.0
- Replacing the Linux 2.4 networking subsystem with the one from uClinux 2.4

One of the more controversial areas of potential code size reductions is in the area of run-time library optimizations. Due to the prevalence of glibc as the defining API for the Linux workstation and server platform, changes away from the use of glibc are feared to pose a fragmentation threat to various embedded Linux systems. The real risk of fragmentation is primarily an issue in case there is a single prevailing API in embedded systems (as in the x86 in the desktop), and less of an issue due to the wide number of embedded CPU architectures. The best strategy for binary application re-use is to utilize a higher-level language such as Java. This would allow for a replacement library for glibc provided that it implements sufficient systems resources to provide dynamic linking of a predominant share of Linux applications that are applicable to embedded devices, including a Java VM. However, the size of the higher-level language may negate the savings in using a C library smaller than glibc.

Due to insufficient business motivation, the implementation of a lower-profile alternative to glibc that would have the equivalent popularity in the embedded market has not taken root. A successful implementation would require a self-funding business model (to justify an investment), open source licensing (to encourage a critical mass of adoption), and potentially, a single or pair of sponsoring entities (to serve as industry leaders in defining a de facto standard).

The following table summarizes various memory footprint optimizations.

Approach	Description
gcc – compiler optimization	Use the built-in compiler code size optimization features. Take careful measurements, as the optimizations may not be pronounced for processors used in embedded devices. Track the increase in execution time and RAM requirements.
xconfig – remove unnecessary features	Understanding which Linux features are required in an embedded device and excluding all other features is the easiest, fastest, and more reliable way to reduce the memory footprint.
Library Optimizer – remove unnecessary library functions	If a significant number of specialty libraries are required, removing all unused functions in those libraries will reduce the memory requirements. If new applications can be loaded, removing library functions can create compatibility problems. Solving these problems usually entails including the needed functions with the new application (which can have licensing implications) or including a full version of the library with the application.
Older kernel	New features are added to each new version of the Linux kernel. Generally, the kernel grows as a result. Embedded devices may not require all the latest features. Using an older kernel, with patches for the known defects, will likely be a memory saving approach. Be careful to understand the limitations of the chosen kernel.
Older kernel subsystem	An older version of a kernel subsystem can be used in place of the version in the embedded device kernel. Since the interfaces between kernel subsystems chance in unpredictable ways as new kernels are released, using an older version of a subsystem can be very difficult. The memory savings advantage is likely to be outweighed by the effort involved with getting the older version working correctly with a newer kernel.
cramfs – file system image	When performance or power issues don't allow execute-in-place, compressing the kernel and the file system is an easy way to reduce the

Approach	Description
compression	ROM/flash memory footprint. Monitor the performance carefully; especially application load times.
Statically link entire code, remove file system	The predominant approach to embedded software is to create one large executable file. This approach is obviously not very flexible for enhancements once a product shipped. However, if the embedded device will not require field upgrades, linking the entire code set into one executable will reduce the memory footprint. Be careful to understand the license for each code set included in the final link.
Header file tuning parameters	Many of the kernel header files contain tuning parameters. Once the embedded device is operating reliably, adjusting the tuning parameters may improve performance or reduce memory requirements.
Static buffer / array analysis	Using the `size`, `nm`, or `objdump` utilities, the statically allocated memory requirements can be monitored. Identifying the largest static memory users is easy, giving the developer hints on where to start looking when tuning RAM memory usage.
Dynamic memory analysis	The `proc` file system allows the dynamic memory usage to be monitored. The `proc` file system can even be enhanced to provide different views of who is using memory and in what way. Large dynamic memory users can then be examined to see what options exist to reduce usage.

Table 32: Memory footprint reduction options summary

9 Appendix A - Software Documentation

9.1 Linux Utility Commands

The information used to document the Linux utility commands used in this case study is from the Linux man pages and the Linux info system.

9.1.1 dmesg

Prints or controls the kernel ring buffer. The kernel ring buffer contains the end of the various kernel generated diagnostic messages.

9.1.2 du

du reports the amount of disk space used by the specified files and/or directories. Normally the disk space is printed in units of 1024 bytes.

9.1.3 nm

A	The symbol's value is absolute, and will not be changed by further linking.
B	The symbol is in the uninitialized data section (known as BSS).
C	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references. For more details on common symbols, see the discussion of -warn-common in *Note Linker options: (ld.info)Options.
D	The symbol is in the initialized data section.
G	The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.
I	The symbol is an indirect reference to another symbol. This is a GNU extension to the a.out object file format which is rarely used.
N	The symbol is a debugging symbol.
R	The symbol is in a read only data section.
S	The symbol is in an uninitialized data section for small objects.
T	The symbol is in the text (code) section.
U	The symbol is undefined.
V	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol

	becomes zero with no error.
W	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

9.1.4 size

size lists the section sizes--and the total size--for each of the object or archive. The output is a table, one row for each file, and the columns described in the following table.

Column Title	Meaning
text	The symbol is in the text (code) section.
data	The symbol is in the initialized data section.
bss	The symbol is in the uninitialized data section (known as BSS).
dec	The total of text, data, and bss, expressed in decimal format.
hex	The total of text, data, and bss, expressed in hexadecimal format.
filename	The name of the file

10 Appendix B - GStreamer Documentation

From the application programmer's point-of-view, GStreamer consists of a small set of objects, as shown in Figure 8. Each object is described below.

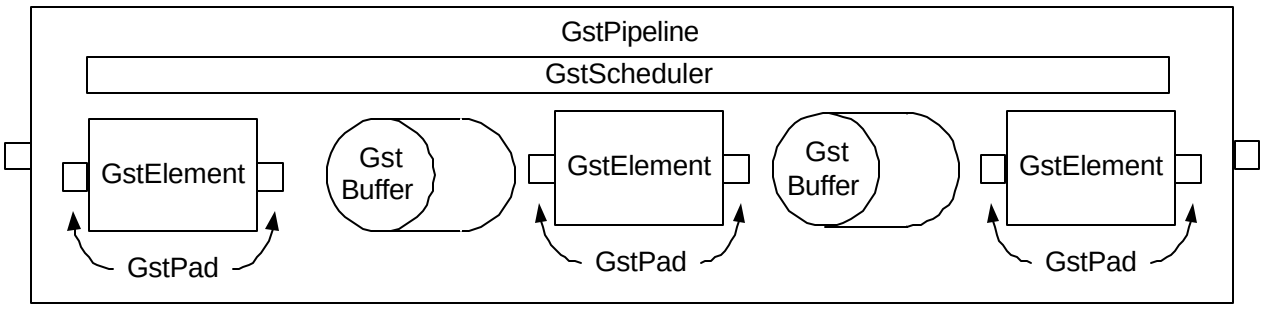

Figure 8: GStreamer Example Pipeline

10.1 GstElement

`GstElement` is the fundamental object of a pipeline. It contains information about the external connections, parentage, and the internal functions necessary for the element to do its job. Parentage is handled by the parent pointer in the base `GstObject`. External connections are handled as a list of `GstPads`. Function pointers keep track of the necessary internal routines.

Each element has a current and pending state, which is one of NULL, READY, PAUSED, or PLAYING. This state is used to determine whether the element should, for instance, have the source file open or not. The higher-level scheduler is responsible for actually running the plugin in the PLAYING state, so the plugin isn't responsible for determining when to run.

```
gst_element_add_pad(GstElement *element, GstPad *);
gst_element_remove_pad(GstElement *element, GstPad *);
```

Add pads to an element or remove them.

```
GstPad *gst_element_get_pad(GstElement *element, gchar *padname);
```

Retrieve a pad from the element by name.

```
gst_element_set_state(GstElement *element, GstElementState state);
```

Attempt to set the state of the element.

```
gst_element_connect(GstElement *srcelement, gchar *srcpadname,
                    GstElement *sinkelement, gchar *sinkpadname);
```

Convenience function to connect two pads from different elements by name.

10.2 GstPad

A pad is a connection point for an element. Pads are either source or sink pads, and pads can only connect with pads of the opposite type. Data always flows from a source pad to a sink pad.

The pad optionally (depending on the operating mode of the element) contains function pointers to the routines that will do the processing work for the element, as well as function pointers the scheduler uses to plan out the execution sequence.

`gst_pad_set_<*>_function(GstPad *pad GstPad<*>Function *<*>);`
Set the <*> function for the pad,
<*> can be one of `chain`, `get`, `event`, `negotiate`, `newcaps`, or `bufferpool`.

`gst_pad_connect(GstPad *srcpad, GstPad *sinkpad);`
Connect two pads together.

10.3 GstBin

A bin is a special case of a `GstElement` that contains other elements. The default bin is nothing but a simple container, but there are other bins with various features, such as `GstThread`, which puts its children into a separate `pthread`. Functions are provided to manage adding and removing elements.

```
gst_bin_add(GstBin *bin, GstElement *element);
gst_bin_remove(GstBin *bin, GstElement *element);
```
Add elements to a bin or remove them.

`gst_bin_iterate(GstBin *bin);`
Force execution of all the elements in the bin (if it has a scheduler).

10.4 GstScheduler

Certain types of bins (like `GstPipeline` and `GstThread`) have an associated scheduler that plans for the execution of the elements under its control. It uses function pointers located in the elements and pads to implement the plan.

All the scheduler methods are private to the core library, and only need to be used when writing a new scheduler.

10.5 GstBuffer

The heart of GStreamer's media-handling is the `GstBuffer`. It has a pointer to a memory region, as well as metadata such as the size, offset, and timestamp of the buffer. They also have reference counts and locks to ensure correctness.

Buffers can be children of other buffers, with their data pointer residing within the memory region of the parent buffer. The parent is not freed until all the children are freed.

Buffers can be merged together, either by creating a copy of the data, or if the buffers are contiguous with the same parent buffer, a new child of that parent can be create to span the buffers. Combined with elements that are written to optimize for this, almost total zero-copy can be achieved.

Buffer pools allow for the creation and management of specialized buffers, such as audio or video DMA buffers. These buffer pools can be acquired from further downstream in the pipeline to avoid

copies (i.e. decoded video can be written directly to frame buffer memory even through intervening filters).

Copy-on-write will be implemented at some point in the near future, to ensure that branched pipelines will not cause elements to simultaneously rewrite the same buffer.

`GstBuffer *gst_buffer_new();`
Create a new buffer with no pointers or metadata.

`GstBuffer *gst_buffer_new_from_pool(GstBufferPool *pool, int offset, int size);`
Create a new buffer with a data pointer and metadata from a buffer pool.

`gst_buffer_ref(GstBuffer *buf); gst_buffer_unref(GstBuffer *buf);`
Handle reference counting.

`GstBuffer *gst_buffer_create_sub(GstBuffer *buf, int offset, int size);`
Create a child buffer from the middle of a parent buffer.

```
GstBuffer *gst_buffer_span(GstBuffer *buf1, int offset, GstBuffer *buf2,
                           int size);
```
Create a new buffer that spans the data areas of buf1 and buf2.
A child buffer of a shared parent will be created if possible, otherwise data is copied.

10.6 GstCaps/GstProps

All pads have `GstCaps` capabilities associated with them, which describes the data expected or produced on that pad. The properties consist of a list of tag/type/value tuples, describing such things as bit rate, width/height, color format, etc. A `GstCaps` structure associates a list of these lists with a MIME type, and these structures form a list. When attached to a `GstPad`, they provide a means of verifying that the data types are compatible, as well as working out a common format.

When pads are connected, they engage in a negotiation process where common ground is found. A default is provided that simply looks at the capabilities template provided by the plugin, but the element can provide an override function for each pad to negotiate in a more intelligent manner.

Most of the capabilities handling in code is done using constructor macros, but code must also be able to create and inspect capability live, in order to do negotiation at connect time:

```
void gst_caps_set_props(GstCaps *caps, GstProps *props);
GstProps *gst_caps_get_props(GstCaps *caps);
```
Set and retrieve the properties pointer of a `GstCaps` object.

`void gst_caps_set(GstCaps *caps, gchar name, arg...);`
Set properties in a `GstCaps`.

`gst_caps_get_<*>(GstCaps *caps, gchar name);`
Retrieve a specific type of property from the caps by name.
<*> can be one of `int`, `float`, `fourcc`, `boolean`, or `string`.

10.7 GstPlugin

A plug-in is a single shared library that contains a collection of plug-ins and other loadable objects. A `GstPluginFeature` can be a `GstElementFactory` (used to create instances of elements), or a `GstType`, which is an abstraction which helps to manage basic media types. Plug-ins can be found via any of several methods, including a system-wide XML registry, searching the a path, and absolute path loading.

When a plug-in is loaded, a `GstPluginDesc` structure is searched by the name `plugin_desc`, which provides the version of GStreamer the plug-in was built against, the name, and the initialization function, which is called immediately.

`gst_plugin_add_feature(GstPlugin *plugin, GstPluginFeature *feature);`

Add a plug-in feature to the list in the plug-in.

11 Appendix C - Case Study Starting Configuration

The .config file in the case study source code Linux subdirectory lists which options are included / excluded. The case study .config file lines which include functionality are listed below. The xconfig Linux kernel configuration tool can be used to change the .config file (as explained in the *xconfig – Feature Inclusion Configuration Tool* section).

```
# Automatically generated make
# config: don't edit
# Edited – all lines with “is not
# set” have been removed,
# as well as blank lines.
CONFIG_ARM=y
CONFIG_UID16=y
#
# Code maturity level options
#
CONFIG_EXPERIMENTAL=y
#
# Loadable module support
#
CONFIG_MODULES=y
#
# System Type
#
CONFIG_ARCH_OMAP=y
#
# TI OMAP Implementations
#
CONFIG_ARCH_OMAP1510=y
CONFIG_CPU_32=y
#
# Processor Type
#
CONFIG_CPU_32v4=y
CONFIG_CPU_ARM925=y
CONFIG_NET=y
CONFIG_CPU_ARM925_CPU_IDLE=y
CONFIG_CPU_ARM925_I_CACHE_ON=y
#
# General setup
#
CONFIG_NET=y
CONFIG_NWFPE=y
CONFIG_KCORE_ELF=y
CONFIG_BINFMT_ELF=y
CONFIG_LEDS=y
CONFIG_LEDS_TIMER=y
CONFIG_LEDS_CPU=y
#
# TI ARM925 CPU CONFIG
#
CONFIG_CPU_ARM925_NON_STREAMING_ON=y
#
# OMAP1510 Support
#
CONFIG_DSPLINUX_OMAP1510_SERIAL=y
CONFIG_DSPLINUX_OMAP1510_SERIAL_CONSO
LE=y
CONFIG_DSPLINUX_OMAP1510_HWERROR=m
#
# Input core support
#
CONFIG_INPUT=y
CONFIG_INPUT_KEYBDEV=y
CONFIG_DSPLINUX=y
#
# Memory Technology Devices (MTD)
#
CONFIG_MTD=y
#
# MTD drivers for mapped chips
#
CONFIG_MTD_CFI=y
CONFIG_MTD_CFI_INTELEXT=y
#
# User modules and translation layers
for MTD devices
#
CONFIG_MTD_BLOCK=y
#
# Block devices
#
CONFIG_BLK_DEV_RAM=y
CONFIG_BLK_DEV_RAM_SIZE=4096
CONFIG_BLK_DEV_INITRD=y
#
# Networking options
#
CONFIG_UNIX=y
CONFIG_INET=y
CONFIG_IP_PNP=y
CONFIG_IP_PNP_BOOTP=y
CONFIG_IP_PNP_RARP=y
#
# Network device support
#
```

```
CONFIG_NETDEVICES=y
#
# Ethernet (10 or 100Mbit)
#
CONFIG_NET_ETHERNET=y
#
# Character devices
#
CONFIG_VT=y
CONFIG_VT_CONSOLE=y
#
# File systems
#
CONFIG_ISO9660_FS=y
CONFIG_JOLIET=y
CONFIG_PROC_FS=y
CONFIG_DEVFS_FS=y
CONFIG_EXT2_FS=y
#
# Network File Systems
#
CONFIG_NFS_FS=y
CONFIG_NFS_V3=y
CONFIG_ROOT_NFS=y
CONFIG_SUNRPC=y
CONFIG_LOCKD=y
CONFIG_LOCKD_V4=y
#
# Partition Types
#
CONFIG_MSDOS_PARTITION=y
CONFIG_NLS=y
#
# Native Language Support
#
CONFIG_NLS_DEFAULT="iso8859-1"
CONFIG_NLS_CODEPAGE_437=y
CONFIG_NLS_CODEPAGE_850=y
CONFIG_NLS_ISO8859_1=y
#
# Console drivers
#
CONFIG_FB=y
#
# Frame-buffer support
#
CONFIG_FB=y
CONFIG_DUMMY_CONSOLE=y
CONFIG_FBCON_FONTS=y
CONFIG_FONT_8x8=y
CONFIG_FONT_ACORN_8x8=y
#
# Input core support
#
CONFIG_INPUT=y
CONFIG_INPUT_KEYBDEV=y
#
# Kernel hacking
#
CONFIG_FRAME_POINTER=y
CONFIG_DEBUG_ERRORS=y
CONFIG_DEBUG_USER=y
CONFIG_DEBUG_LL=y
```

12 Acknowledgements

12.1 libwww Copyright Notice

libwww: W3C's implementation of HTTP can be found at: http://www.w3.org/Library/
Copyright © 1994-2000 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. This program is distributed under the W3C's Software Intellectual Property License. This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the imp lied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See W3C License http://www.w3.org/Consortium/Legal/ for more details.

Copyright © 1995 CERN. "This product includes computer software created and made available by CERN. This acknowledgment shall be mentioned in full in any product which includes the CERN computer software included herein or parts thereof."

12.2 W3C® SOFTWARE NOTICE AND LICENSE

Copyright © 1994-2001 World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/

This W3C work (including software, documents, or other related items) is being provided by the copyright holders under the following license. By obtaining, using and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:
Permission to use, copy, modify, and distribute this software and its documentation, with or without modification, for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the software and documentation or portions thereof, including modifications, that you make:

1. The full text of this NOTICE in a location viewable to users of the redistributed or derivative work.
2. Any pre-existing intellectual property disclaimers, notices, or terms and conditions. If none exist, a short notice of the following form (hypertext is preferred, text is permitted) should be used within the body of any redistributed or derivative code: "Copyright © [$date-of-software] World Wide Web Consortium, (Massachusetts Institute of Technology, Institut National de Recherche en Informatique et en Automatique, Keio University). All Rights Reserved. http://www.w3.org/Consortium/Legal/"
3. Notice of any changes or modifications to the W3C files, including the date changes were made. (We recommend you provide URIs to the location from which the code is derived.)

THIS SOFTWARE AND DOCUMENTATION IS PROVIDED "AS IS," AND COPYRIGHT HOLDERS MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
COPYRIGHT HOLDERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SOFTWARE OR DOCUMENTATION.
The name and trademarks of copyright holders may NOT be used in advertising or publicity pertaining to the software without specific, written prior permission. Title to copyright in this software and any associated documentation will at all times remain with copyright holders.

12.3 MPG123 License

Copyright (c) 1995-99 by Michael Hipp, all rights reserved. Parts of the software are contributed by other people, please refer to the README file for details!