THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Native OKL4 Web Browser

Josh Matthews

Bachelor of Engineering (Software Engineering)

Supervisor: Gernot Heiser

Assessor: Ihor Kuz

June 1, 2010

Abstract

The rapid growth of both the mobile and cloud computing industries heralds a trend towards the convergence of embedded and web technologies. While the client-side of today's mobile ecosystem is dominated by rich operating systems and relatively heavyweight application stacks, this trend raises the possibility of utilizing only web technologies on the client. Palm's WebOS and Google's Chrome OS are proof of the viability of such an approach.

However, this approach has major security implications; the web is inherently a hostile environment. Recent advances in secure browser architectures have relied on operating system fundamentals and mechanisms, such as the separation of each web application into its own process and the creation of centralized "browser kernels”. Such techniques, in practice, still depend on an underlying rich OS for their implementation, with the resultant impact on security posture: the architecture is only as secure as the OS.

An alternative approach is to implement the browser architecture on a microkernel, which has the potential to provide two advantages: the minimization of the trusted computing base (TCB), and the ability to strongly isolate components of the browser architecture by utilizing underlying security primitives (such as capabilities) in the implementation.

This thesis provides the fundamental basis for the investigation of this concept. We port the WebKit browser architecture to OKL4 4.0, a third-generation microkernel that provides a capability-based security architecture, strong isolation between subsystems, and a minimal footprint. We prove the viability of the approach by displaying basic web pages on the implementation, using the Beagle Board development platform.

Acknowledgements

I would like to thank Dr. Gernot Heiser for his continuing support and encouragement, in this and all endeavors.

I would also like to thank my wife, Lauren, for sticking by me through an eleven-year degree!

Contents

Chapter 1

Introduction

1.1 Motivation

With the simultaneous growth of both cloud computing and web connected embedded devices, an opportunity has arisen that will enable the replacement of heavyweight client side application stacks (such as Android) with lighter-weight web-based architectures. It is certainly possible, particularly with the improvements in client-side application development brought on by HTML5, to construct an embedded device using only an embedded web browser to provide the entire UI and application stack. Indeed, projects like Chrome OS and WebOS have already made significant advancements towards this vision. However, such architectures have obvious security concerns: web browsers are often susceptible to attacks.

What is needed is a secure browser architecture that can prevent such attacks, and can limit the impact of attacks when they do occur. This is a common use case for microkernel-based architectures: minimization of the trusted computing base (TCB), and ability to strongly isolate system components using some underlying security primitive (such as capabilities). Additionally, browsers engines themselves are beginning to represent operating systems, with secure browser architectures that treat the browser engine as a multi-principled OS having success in implementing browser security policies [1].

The main motivation for this thesis is the premise that a microkernel, such as OKL4, forms an ideal basis on which a secure browser architecture can be implemented.

1.2 Goals

The primary goal of this thesis is to port the WebKit browser engine to an OKL4 4.0 Lightweight Execution Environment (LWEE) on the Beagle Board platform, to the point where it will be able to display basic web pages. This system will provide the groundwork upon which future work will be able to build; in particular, it will port enough of WebKit so that investigations into separating components of the architecture into OKL4 cells, perhaps utilizing OKL4 capabilities for security policy implementation, will be able to begin. To achieve this goal, we need to:

- Develop an OKL4 LWEE OS personality to support the fundamental OS dependencies of WebKit (e.g. threads).
- Develop or port all library dependencies of WebKit to an OKL4 LWEE programming environment (e.g. graphics libraries).
- Develop or port all drivers required by those libraries (e.g. graphics drivers).
- Port WebKit to utilize the developed LWEE OS personality and programming environment.

The success of this effort will be determined by the ability of the implementation to display a basic web page, composed of HTML, CSS, and images, over an internal network. JavaScript execution, user input (e.g. keyboard or mouse), and plug-in support (e.g. Flash) will not be considered.

1.3 Thesis Structure

Chapter 2: Background

This chapter provides an overview of the WebKit browser engine, the Origyn Web Browser Abstraction Layer modifications made to WebKit, the OKL4 4.0 Microvisor, and the Beagle Board development platform. This provides necessary background information for the rest of the thesis.

Chapter 3: Approach

This chapter describes the high-level approach taken to achieve the goals of the thesis. The requirements for and architecture the OS personality, supporting libraries, and drivers required to support WebKit are detailed.

Chapter 4: Implementation

This chapter describes the low-level implementation of the OS personality, supporting libraries, and drivers required to support WebKit, including design tradeoffs, issues faced, and challenges overcome.

Chapter 5: Evaluation

This chapter presents a brief evaluation of the completed work, detailing success criteria achieved and presenting some basic performance analysis.

Chapter 6: Future Work

This chapter discusses the shortcomings of the thesis implementation and proposes future work.

Appendices

- Appendix A provides a listing of the system configuration file used to configure the completed system.
- Appendix B provides a listing of the HTML file used to test the system.

Chapter 2

Background

2.1 WebKit

WebKit [2] is an open source web browser engine, originally developed as a fork of the KDE KHTML html renderer and KJS JavaScript engine projects. Widely used, WebKit is the engine for several popular browsers including Apple Safari and Google Chrome. Written almost entirely in C++, The WebKit architecture is composed of three core components: JavaScriptCore, WebCore, and WebKit.

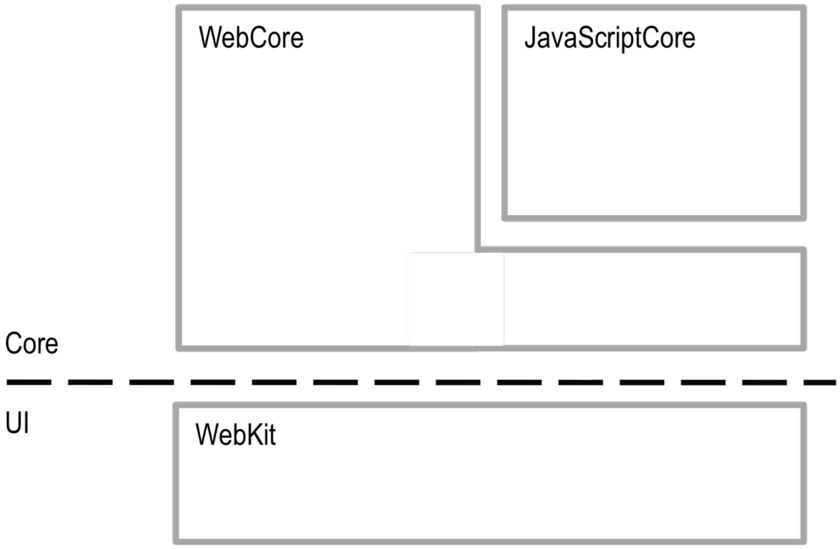


Figure 2.1: WebKit Architecture

JavaScriptCore

JavaScriptCore is the JavaScript execution engine of WebKit, containing the JavaScript parser, bytecode compiler, profiler, and runtime. It also contains the wtf ("Web Template Foundation") utility library consisting of various helper classes (such as a malloc implementation) and common data structures and abstractions (such as a HashMap implementation and a threading abstraction), which is utilized by other components in WebKit.

JavaScript core is relatively cross-platform, with the minimal platform dependencies that do exist (primarily threading and memory management) abstracted in the wtf library.

WebCore

WebCore is the main engine of WebKit, responsible for HTML and XML parsing and page rendering, network operations, CSS, SVG, and DOM support, accessibility, editing, internationalization, and history management, and client-side storage and plug-in support. WebCore is monolithic (with a number of interdependencies between the above responsibilities), and highly platform dependent in terms of graphics, network, threading, and storage operations.

WebKit

WebKit is the front-end layer utilized by the WebKit architecture to present the user interface. Entirely platform-dependent, WebKit is responsible for loading the UI and handling common user interaction tasks, such as navigation.

2.2 Origyn Web Browser Abstraction Layer

While attempts have been made in the WebKit architecture to abstract platform dependencies, the monolithic nature of WebCore, in practice, renders the task of porting the architecture to new platforms quite cumbersome. In particular, platform dependencies are routinely conditionally compiled in throughout the code base without specific documentation. Additionally, a "platform" is usually composed of some combination of graphics, network, and threading components; the standard build system for WebKit (Bakefile), while allowing the developer to select a platform as a target, generally makes it difficult to individually select components of an existing platform to utilize in a new port (for example, re-using the pthread library support of a Linux platform port is difficult).

The Origyn Web Browser (OWB) project [3] was created to overcome these deficiencies, with a specific objective of being utilized in consumer electronics (CE) ports of WebKit. OWB introduces the concept of a Browser Abstraction Layer (BAL), which is essentially a refactoring of WebKit to specifically isolate individual components of platform dependencies (graphics, network, and threading, for example) into a C++ class inheritance hierarchy.

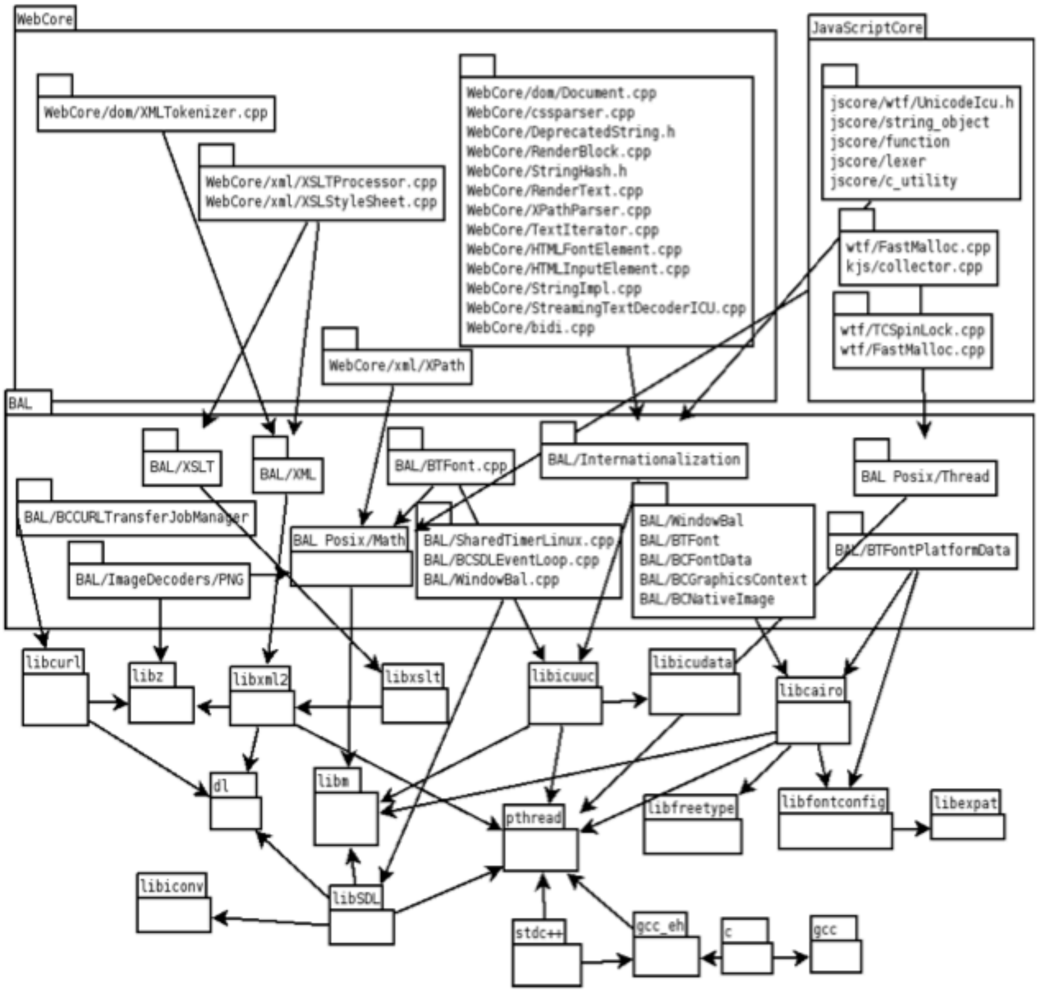


Figure 2.2: OWB BAL Architecture

OWB also introduces a new build system (CCMake), modified to be aware of all BAL components, which allows the developer to select individual aspects of different ports and combine them to create a new platform port. This is particularly useful for embedded platform porting, in which it is common to use, for example, only a subset of POSIX support combined with some custom libraries; OWB renders these components "pluggable".

A further advantage of OWB is existing platform support for common embedded libraries, such as the Simple Direct Media Layer (SDL) graphics library [4]. The WebKit project, as it is primarily intended for desktop use, generally lacks support for these libraries (in particular, SDL is not supported by the WebKit project).

2.3 The OKL4 Microvisor

OKL4 4.0 (the "OKL4 Microvisor") is a third-generation microkernel that allows a combination of both virtualization and fine-grained componentization in embedded system design and implementation. With a small static memory footprint (approximately 45 KiB), OKL4 enables system components to be strongly isolated in partitions known as "cells".

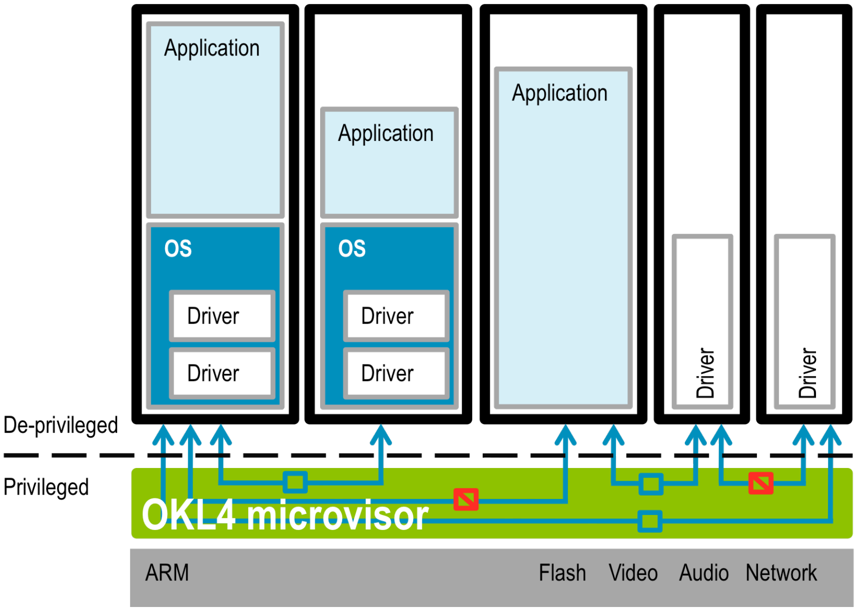


Figure 2.3: OKL4 & Secure HyperCell Technology

OKL4's primary responsibility is as a hardware resource manager: it provides virtual hardware abstractions, consisting of virtual CPUs (vCPUs), virtual MMUs (vMMUs), and virtual device abstractions (virtual interrupts and memory mapped register sets), to cell environments. There are two types of cell environments:

- *Virtual Machine Environments*: VM environments are used to virtualize existing operating systems and utilize all hardware abstractions.

- *Lightweight Execution Environments*: LWEE environments are used to host fine-grained components of the system, such as individual applications or drivers, without requiring an underlying operating system to support the component. An LWEE is "lightweight" because it does not utilize a vMMU, and hence avoids any memory overhead of e.g. shadow page tables. LWEE's contain a *programming environment*, which consist of any supporting libraries required by the application or driver. A programming environment may also contain an *OS personality*, consisting of a set of libraries that provide underlying OS primitives (such as threading, mutexes, and semaphores) that may be required.

Additionally, OKL4 facilitates communication between cells by providing virtual communication mechanisms. A "channels" primitive is provided which allows general purpose, per-channel-configurable fixed-size message passing utilizing a dual-buffer single-copy approach. Configurable virtual interrupt lines can also be defined, allowing a cell to raise a vIRQ to another cell.

Security, in particular strong isolation between cells, is facilitated by a capability-based mechanism. Capabilities are vCPU-scoped tokens that convey both identification over system objects and privileges to perform some subset of available operations on the identified object. Capabilities are used as the allocation mechanism for all system objects (such as vCPUs and channels) to cells; a cell cannot utilize (indeed, cannot even refer to) a system object unless it has received a capability to it. As they are locally scoped, the ownership of a capability cannot be transferred between cells. Capabilities are allocated by the system designer at system configuration time, enabling very fine-grained control over the security partitioning of the system.

OKL4 is distributed as a Software Development Kit (SDK), which contains:

- Binary releases of the OKL4 kernel and system header files.
- The *ok* tool, which takes as input an XML description of the system configuration, along with binaries for each cell, and *merges* those inputs into a bootable system image (ensuring, for example, capability allocation as specified by the system configuration).
- ISA, System on Chip and Board support for particular processors, SoCs, and boards (e.g. ARM Cortex-A8, OMAP3530, and the Beagle Board board).
- A set of included *packages* (in source and binary form), including example OS personalities and pre-ported libraries and drivers, which enable the system developer to define an initially programming environment for a LWEE.
- A virtual serial console mechanism, such that multiple cells can perform serial debugging output to virtual consoles on a host machine.

2.4 The Beagle Board Platform

The Beagle Board [5] is an embedded development platform with the following specification:

- Processor: ARM Cortex-A8
- SoC: TI OMAP3530
- DRAM: 256 MiB
- NAND: 256 MiB
- Display: DVI-D out
- Ports: Serial, USB OTG, USB Host
- Storage: MMC/SD card slot

Chapter 3

Approach

3.1 Overview

The high-level approach to the thesis is to develop an OKL4 LWEE OS personality that supports the fundamental OS requirements of the WebKit architecture, port the major supporting libraries required by WebKit and the OS personality, and develop or port OKL4-based drivers required by those libraries.

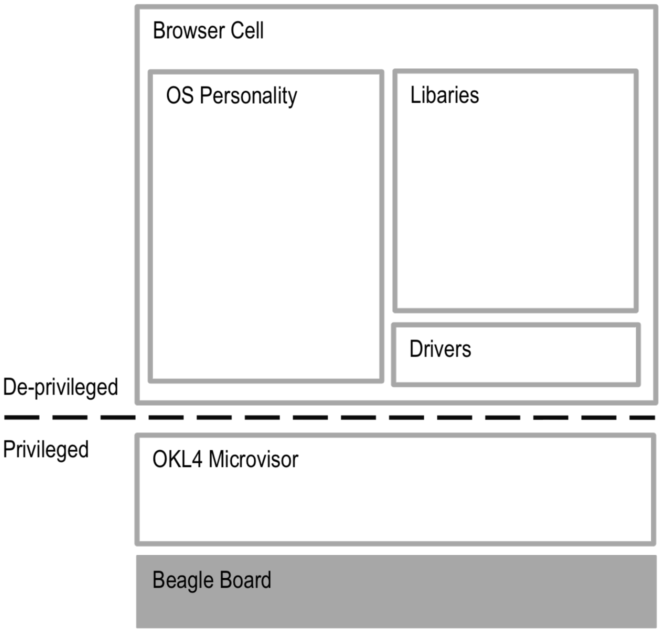


Figure 3.1: High Level Architecture

The design is intentionally monolithic (i.e. uses a single OKL4 LWEE cell, which we will refer to as the "Browser cell") to ease the porting process for this initial work. As described in Section 1.1 (Motivation), future work will be able to build on this design by separating components of the architecture into multiple LWEE cells.

Important to note is that while the stated goals of the thesis do not include JavaScript execution, it is necessary to perform a port of JavaScriptCore because of the wtf library that it contains, which is heavily utilized by WebCore. As the platform dependencies of JavaScriptCore are minimal this was expected to be fairly straightforward, and it was considered easier to port JavaScriptCore than to try to separate the wtf library.

3.2 OS Personality

The purpose of the OS personality is to provide the fundamental supporting OS mechanisms and primitives on which WebKit depends. While the WebKit architecture, particularly with the OWB/BAL mechanism, is somewhat flexible in terms of its OS dependencies, it does contain a core set of requirements that must be provided by the OS. These requirements, and the selected design and methodology chosen to implement them, are outlined below.

3.2.1 Threading

The core requirements of the threading mechanism to be provided to WebKit are:

- Ability to create and delete threads.
- Pre-emptive, timeslice-based scheduling. No priorities are required.
- Ability to obtain details of the thread post-creation, such as the base of the thread's stack.

The BAL allows selection of the POSIX pthread library to implement these threading requirements (consisting of the headers `pthread.h` and `pthread_np.h`. The latter allows querying of pthread parameters post-creation). We determined that the most straightforward approach would be to provide an implementation of pthreads as part of our OS personality.

The OKL4 SDK comes with an example OS personality, OK Real Time Executive No. 2 (okrtx2), which contains support for a thread abstraction and a scheduler. However, the okrtx2 scheduler is cooperatively driven: a runnable thread is executed when the running thread blocks on a synchronization primitive. We determined that a viable approach would be to add preemptive scheduling support to okrtx2, and then layer a pthread compatibility library on top (with pthread functions calling their equivalent okrtx2 functions).

3.2.2 Synchronization Primitives

The core requirements of synchronization primitives to be provided to WebKit are:

- Mutexes.
- Reader-Writer locks.

The BAL allows selection of the POSIX pthread library to implement these synchronization requirements (via primitives `pthread_mutex_t and pthread_rwlock_t)`. As with the threading requirements, we determined that we would implement these pthread primitives.

okrtx2 already contains support for mutexes. We determined that a viable approach would be to add support for reader-writer locks to okrtx2, and develop our pthread compatibility library to utilize the mutex and rwlock primitives provided by okrtx2.

3.2.3 Timers

The core requirements of timing primitives to be provided to WebKit are:

- Ability to program a one-shot timer, to microsecond resolution.
- Ability to pause a thread (sleep), to both second and microsecond resolution.
- Ability to get the current time of day, to microsecond resolution.

The BAL provides a timer abstraction with various implementations. We determined that it would be straightforward to modify the Linux timer implementation to utilize an OKL4 timer driver (the approach for which is discussed in Section 3.4.1 [Timer Driver]). Additionally, the BAL allows a configuration to use the libc-standard `sleep` and `usleep` functions to implement the sleep requirements and `gettimeofday` for the current time; we determined that a viable approach would be to implement these functions in our OS personality.

3.2.4 File System

The core WebKit file system requirements are:

- Ability to create, open, read, write, seek, and close regular files.
- Ability to create, open, read, and close directories.

The BAL allows selection of the standard libc IO functions (`open`, `read`, `write`, etc.) to implement file operations. Additionally, it allows selection of the POSIX-standard `dirent.h` header to implement directory operations.

The OKL4 SDK comes with an implementation of a FAT32 filesystem library. It was determined that a viable approach would be to implement the above standard functions, hooking them up to the provide FAT32 filesystem library.

3.2.5 Networking

The core WebKit networking requirements are:

- Ability to perform all HTTP operations (include request/response, HTTP keepalives, etc.)

The BAL allows selection of the Curl library [6] to implement this required functionality. Curl requires an underlying TCP/IP stack to operate; the OKL4 SDK comes with an implementation of the Lightweight IP Library (lwip) [7], which provides a TCP/IP stack. Curl also requires an implementation of a semaphores synchronization primitive, with the standard create, semv, semp, and delete operations.

It was determined that a viable approach would be to port the Curl library to our OS personality. We will develop a semaphore primitive as an addition to okrtx2, and hook up Curl to utilize this primitive along with the provided LwIP library.

3.2.6 Graphics

The core WebKit graphics requirements are:

- Ability to create and destroy graphics surfaces.
- Ability to draw rectangles and paths on the surface, along with the ability to fill regions with a color, pattern, or gradient.
- Ability to display jpeg and png images on the graphics surface.

The BAL allows selection of the Simple Direct Media Layer (SDL) library (with the addition of a supporting graphics library, SDL_gfx) to implement these graphics requirements. It was determined that a viable approach would be port both the SDL and SDL_gfx libraries to execute in our OS personality.

A further advantage of using the BAL to assist in embedded WebKit porting is that it includes an embedded font library, obviating the need to port our own font library. This embedded font library is fixed-space, fixed-size only, but is sufficient for use in our initial port and requires no effort on our part to utilize.

3.3 Libraries

In addition to the above libraries that will form part of our OS personality, the following libraries are also required by WebKit:

- Image libraries: libjpeg and libpng (include the supporting zlib library).
- XML parsing library: libxml

The OKL4 SDK already provides a package supporting libjpeg. It was determined that a viable approach would be to port the remaining libraries to our LWEE programming environment.

3.4 Drivers

In order to support the above OS personality and programming environment, several drivers need to be available in our system. The requirements of these drivers, along with the chosen approach for implementation, are outlined below.

3.4.1 Timer

To support the addition of pre-emptive scheduling to okrtx2, along with the timer functionality required by WebKit, a timer driver with microsecond resolution and the ability to program multiple one-shot and periodic timers is required.

The OKL4 SDK is distributed with a driver for the general purpose timers (GPTs) on the OMAP3530 SoC (on which there are 12 GPTs in total). Additionally, the OKL4 SDK provides a package, *libtimeout*, which facilitates the programming of multiple virtual timers over a single hardware timer. The combination of this driver and library completely fulfill our timer driver requirements.

3.4.2 Secondary Storage

To support the file system requirements of WebKit, and the dependencies of the FAT32 file system library to be used, a driver for a secondary storage device is required. The OKL4 SDK is distributed with a driver for the NAND device on the OMAP3530 SoC, provided via two packages, *libftl* (a Flash Translation Layer driver) and *libnand* (the actual NAND driver). Together, these packages completely fulfill our storage driver requirements.

3.4.3 Network

To support the networking requirements of WebKit, a network device driver is required. This is troublesome on the Beagle Board platform: the board itself has no included network device. An option is to utilize a USB-Ethernet dongle attached to the USB-host connection of the Beagle Board. However, to support a driver for this within our LWEE, a full USB stack would also need to be supported; this is not included in the OKL4 SDK and would be a substantial effort to develop. Fortunately, the OKL4 SDK does ship with a solution: an Android (Linux-based OS) VM environment is provided (which includes a USB stack and drivers for most USB-Ethernet dongles), along with a virtual Ethernet channel library that allows additional OKL4 LWEE or VM cells to utilize (and share access to) the USB-Ethernet dongle driver within Android.

This solution fulfills our requirements for a network driver. However, to correctly utilize in our system, we will need to strip back Android to remove its requirements on those devices that we intend to make available for the exclusive use of our Browser cell (NAND and Graphics).

3.4.4 Graphics

To support the graphics requirements of WebKit, some form of graphics device driver is required. The OKL4 SDK ships with a framebuffer device driver that utilizes the Display SubSystem (DSS) of the OMAP3530 SoC. This driver completely fulfills our requirements for a graphics device driver.

Chapter 4

Implementation

4.1 Overview

4.1.1 Architecture

The following low-level architecture was implemented:

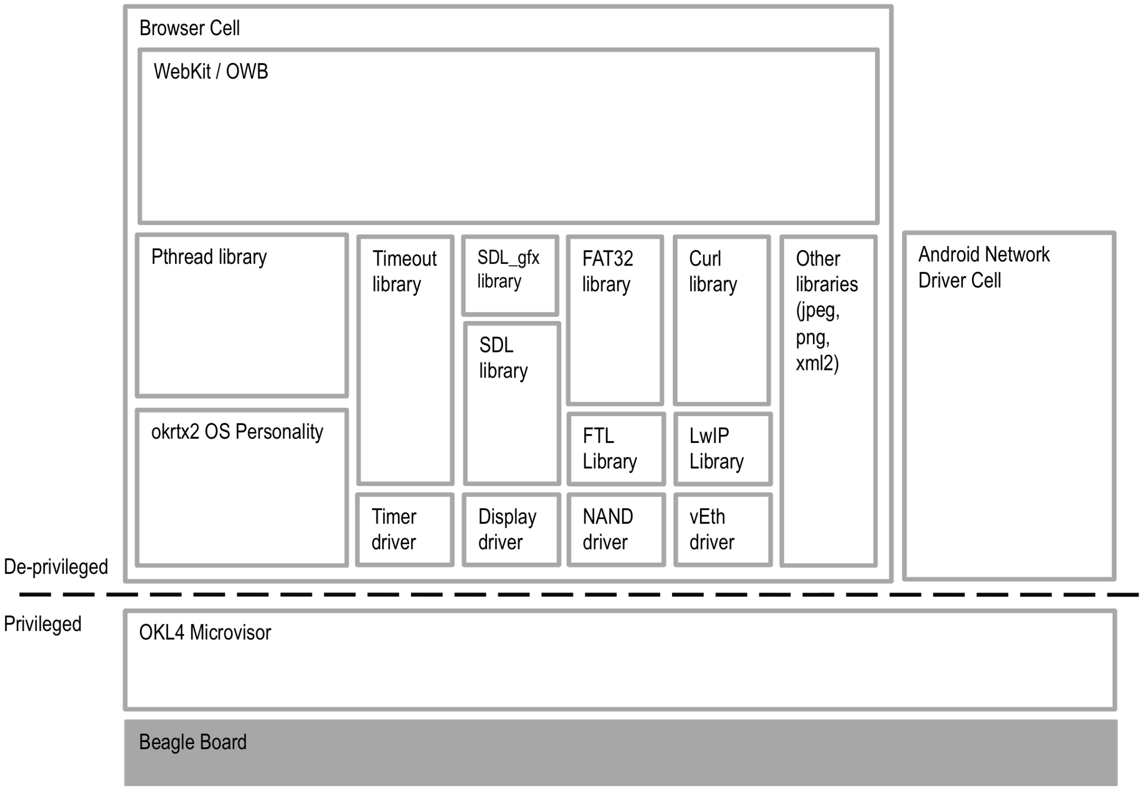


Figure 4.1: Low Level Architecture

4.1.2 Implementation Summary

A summary of the work performed as part of the implementation is as follows:

Component	Functionality	Implementation Required
OS Personality	Threading	• Add pre-emptive scheduling to okrtx2. • Develop pthreads library.
OS Personality	Synchronization	• Develop okrtx2 semaphores. • Develop okrtx2 rw_locks. • Develop pthread rw_locks. • Implement pthread mutexes.
OS Personality	Timer	• Modify BAL Linux timer component to utilize provided timer driver and timeout library. • Implement support for sleep and usleep functions.
OS Personality	File System	• Implement standard libc open/read/write/close functions, hook up to provided FAT32 library. • Implement standard POSIX directory functions, hook up to provided FAT32 library.
OS Personality	Networking	• Port Curl library. • Implement support for provided LwIP library in ported Curl library.
OS Personality	Graphics	• Port SDL library. • Port SDL_gfx library. • Implement support for provided framebuffer driver in ported SDL library.
Libraries	Image libraries	• Port libpng and zlib
Libraries	XML libraries	• Port libxml2
Drivers	Network	• Strip down Android to remove dependency on WebKit allocated devices (NAND and Graphics).
Browser Cell	OWB BAL/WebKit	• Build OWB BAL/WebKit using the above implemented dependencies.

4.1.3 Toolchain and SDK Details

The Code Sourcery 2009-q1 releases [8] of the following toolchains were utilized in the implementation:

- *arm-none-eabi*: used to build all software to execute in the Browser cellw.
- *arm-none-linux-gnueabi:* used to build the Android driver cell.

The 20100325 version of the OKL4 4.0 SDK was used.

4.2 OKRTX2

4.2.1 OKRTX2 Architecture

okrtx2 consists of three packages:

- *okl4_basiccell_rtx2*: this package implements the `_start` function and performs basic OKL4 system initialization, including initializing the vCPU, interrupt vectors, and main system heap. It calls out to initialize the okl4_bascicio_rtx2 package described below, and then calls the `main()` function of the cell. It also implements underlying system utility functions such as `_sbrk`, interrupt management, and system shutdown.

- *okl4_basicio_rtx2*: this package initializes the IO systems, including any required drivers, as well as calls the initialization routine in the okrtx2 package described below. It also provides an implementation of the platform-dependent stub functions called by libc for standard libc functions (such as `_write`).

- *okrtx2:* this package contains the implementation of the okrtx2 OS personality, including threading, the scheduler, mutexes, and basic timed signaling between threads.

4.2.2 Preemptive Scheduling

To implement pre-emptive scheduling, the initialization routine of the okl4_basicio_rtx2 package was modified to program a periodic timeout (using the timeout library) for a configurable duration (10ms was chosen to be the default). The okrtx2 scheduler interface already contained a function to force a reschedule; the timeout handler was programmed to call this function on receipt of the timeout.

4.2.3 Semaphores

Another OKL4 SDK OS Personality package, *okrtx0*, contained an implementation of semaphores. This code was used as a starting point for our implementation of semaphores in okrtx2. The implementation was reasonably straightforward, with simple modifications made to call okrtx2 mechanisms instead of okrtx0 mechanisms. A linked list of semaphore waiters was created by adding a field to the okrtx2 thread structure.

A complication arose in implementing timed waits on a semaphore. okrtx2 already included a timer structure for the sole purpose of raising signals on a particular timeout to a set thread. We extended the timer abstraction to add a *type* field, with two possible types (a signal timer or a semaphore timer). The timer timeout handler was extended to check the type of the timer, and if of the semaphore type, to unblock the thread from its waiting state (if it was still in a waiting state).

4.2.4 Reader-Writer Locks

A reader-writer lock (rwlock) is a particular type of synchronization primitive in which multiple readers can share access to the lock, but only a single writer can hold the lock (to the exclusion of all readers and writers). Reader holders of the lock are *counted*, in that they can request the lock numerous times (but must also perform an equivalent number of unlocks). A writer can only successfully hold the lock once there are no other readers or writers holding the lock. Timed locks are not required.

We developed a from-scratch implementation of rwlocks within the okrtx2 OS personality. We created a rwlock structure to track active readers (threads that have successfully obtained the lock for reading purposes), a single active writer (a thread that has successfully obtained the lock for writing purposes), and blocked readers and writers (threads that have attempted to obtain the lock but have either found an active writer [for readers] or active readers or a writer [for writers]). We then implemented functions to create and delete a rwlock, and to obtain or release both reader and writer locks over the rwlock. Since the lock is intended to service scenarios with many readers and few writers, and to prevent writer starvation, we favored writers over readers when performing an unblock (so that if there were any blocked writers, they would first obtain the lock).

4.3 POSIX

4.3.1 Pthreads

Pthreads is the POSIX standard threading library. A large library with a number of features, pthreads provides:

- Support for the creation, execution, and deletion of threads with configurable control over a number of attributes.
- The ability to join (wait until a thread completes execution) and detach (ensure a thread is deleted immediately upon completing execution) threads.
- Support for guaranteed single execution of a particular function, even if several threads call the function.
- Support for thread local storage, in the form of keys.
- The ability to control a sequence of cleanup operations to be performed by the thread on exiting.
- Support for various synchronization primitives, including mutexes, condition variables, and rwlocks.

The porting of pthreads to a new OS is generally a substantial effort. Fortunately, we had previously undertaken a port of pthreads to a prior version of OKL4 (version 3.0). While the API between OKL4 3.0 and the OKL4 Microvisor (4.0) is substantially and fundamentally different, we were able to reuse much our prior work in our implementation.

In general, the effort was contained to converting OKL4 3.0 primitives to our OS personality (okrtx2) primitives. The major areas of work were:

- Convert OKL4 3.0 threads to okrtx2 threads. This involved further extending our okrtx2 OS personality to store a generalized thread local storage (tls) void pointer in the thread structure, with the ability to set the value of this pointer on thread creation. On creating a new pthread (which then creates a new okrtx2 thread), this pointer is set to back-point to the pthread that has been created. This enables us to discover the currently executing pthread by simply asking the okrtx2 library for the currently executing okrtx2 thread.
- Convert OKL4 3.0 mutex primitives to okrtx2 mutex primitives.
- Create `pthread_rwlock` structures and function implementations, which are simply a wrapper around the new okrtx2 rwlocks.
- Implement `pthread_attr_get_np`, which enables querying of thread attributes (in particular, the stack address) post-thread creation. This was required by WebKit.

- Implement a pthread library initialization routine, which, in particular, initializes a pthread struct in the TLS of the main (currently executing, when called) okrtx2 thread. This function is called from the initialization routine of the okl4_basicio_rtx2 package.

There were two major limitations in our implementation:

- We did not completely implement `pthread_cancel`. Our OKL4 3.0 implementation relied on the use of an OKL4 3.0 mechanism (known as "Exchange Registers") to modify the instruction pointer (IP register) of the thread to be cancelled, if the cancel type was asynchronous (i.e. should occur immediately), so that the thread would immediately jump to a cancellation routine. No equivalent mechanism was present in our OS personality. We could have added a similar mechanism, however this function was only utilized by JavaScriptCore and its correct implementation was not necessary to achieve the goals of the thesis.
- We did not implement `pthread_cond_timedwait`. This would have required, for example, the addition of a third type of timer in okrtx2 (or the creation of some other mechanism). Similar to `pthread_cancel`, this function was only utilized by JavaScriptCore and its correct implementation was not necessary to achieve the goals of the thesis.

4.3.2 Miscellaneous POSIX Functions

In addition to pthreads, a single miscellaneous POSIX function, `posix_memalign`, was required by WebKit to be implemented. This was a simple wrapper around the libc `memalign` function.

4.4 Timers

The BAL provides an abstraction of a single shared timer with a configurable timeout value. On the timeout being reached, a configurable timeout function is called. An implementation is provided for Linux systems; it was straightforward to modify this implementation to utilize the OKL4 SDK timeout library.

Additionally, some minor time functions were required to be implemented, including `sleep`, `usleep`, and `_gettimeofday`. These were straightforward implementations using the okrtx2 timed signaling functions and timeout libraries. These were all implemented in the okl4_basicio_rtx2 package.

4.5 File System

The implementation of the WebKit file system requirements was straightforward. Utilizing the FAT32, FTL, and NAND libraries, the initialization function of okl4_basicio_rtx2 was modified to initialize the NAND driver and mount the FAT32 file system on the NAND device. The FAT32 library already took care of managing open file descriptors, etc. The `_read, _write, _lseek, _fstat, _stat, _open, _close, _unlink, fsync, rmdir, mkdir, and dirname` functions were implemented in okl4_basicio_rtx2 as thin wrappers around the equivalent FAT32 library functions. The only complexity was `_read` and `_write`, which needed to consider whether the given file descriptor referred to stdout, stderr, or stdin and direct those operations to the provided virtual serial console library instead.

4.6 Curl

The Curl networking library, version 7.20.1, was ported to execute on our OS personality. Much of the effort in this implementation was in configuring a minimal and static Curl build to not search standard libraries to obtain implementations of `gethostbyname`, `connect`, `recv`, and `send`. Once this was achieved, the produced curl_config.h header file was manually modified to refer to the LwIP implementations of those functions. An "OKL4" macro was defined, and Curl's `socket.h` header was manually modified to include <lwip/sockets.h> if the OKL4 macro was defined. The only other complexity was that LwIP's netdb.h header file needed to be modified to add the standard conditional inclusion to prevent multiple definition errors (it is unknown why this file did not follow this standard C header convention).

With these changes made, Curl was then compiled successfully. The build failed on attempting to link the curl application, for unknown reasons, but successfully linked the Curl library (libcurl.a), which was all that was necessary for our port.

It should be noted that the option of developing a dummy network library, instead of porting Curl, was initially investigated (since network is not strictly necessary to achieve our initial goals; a dummy network library could simply always return a static html file). A reasonable amount of development time was expended in attempting to get this approach working, with little success – the structure of WebKit networking, even with the BAL abstractions, is quite complex. The approach was eventually abandoned in favor of porting Curl.

4.7 SDL

The SDL graphics library, version 1.2, was ported to execute on our OS personality. The majority of the effort was in developing an SDL video driver for the OKL4 SDK-provided OMAP3530 framebuffer driver.

An `okfb` video driver in the SDL source tree was created. This driver defines a bootstrap structure, `OKFB_bootstrap`, which points to implementations of `OKFB_Available` (which simply always returns true) and `OKFB_CreateDevice` functions. When SDL initializes its video subsystem, it queries this bootstrap structure to determine if the device is available, and, if so, calls the create device function. `OKFB_CreateDevice` creates and returns a standard `SDL_VideoDevice` structure, with initialized function pointers to point to implementations of device initialization and changing video mode settings functions. It also initializes the internal buffer of the device to a statically configured buffer (an OKL4 Virtual Memory item), which is defined in the system configuration file to a size of 2 MiB.

The implemented device initialization function (`OKFB_VideoInit`) simply calls the initialization function in the provided OMAP3530 framebuffer driver, and the set video mode function (`OKFB_SetVideoMode`) calls various OMAP3530 framebuffer driver functions to manipulate the driver settings (such as framebuffer size and format).

Once this was completed, an OKL4 platform was defined to utilize the developed OKFB video driver, and the library was successfully compiled for that platform.

The SDL_gfx graphics utility library, version 2.0.20, was also ported. This provides various graphics drawing functions, such as the ability to draw various geometric shapes on an SDL video surface. This port was straightforward; its only dependencies were on the previously ported SDL library.

4.8 Android Network Driver Cell

As previously described, Android was used to provide the network driver (since it contains an existing USB stack and USB-Ethernet dongle drivers). The primary work to produce this Android Network Driver Cell was in minimizing the configuration of Android so that it does not require the NAND and graphics devices, as these were now allocated for the use by the Browser cell.

This was unfortunately not particularly straightforward, primarily because the OMAP3530/Beagle Board port of the Linux kernel used by Android is somewhat complex.

When the CONFIG_OMAP2_DSS configuration option (used to control the display sub-system) was undefined, the Linux kernel failed to boot. The problem was eventually traced to powerdomains and clockdomains not being initialized if this configuration option was not defined. Removing this initialization condition resolved the issue (no further investigation into the reason for the problem was performed). Once this was done, Android then booted to a virtual console without requiring access to graphics and NAND hardware.

The Android package provided with the OKL4 SDK already provided support for the virtual Ethernet network library (the `eth_channel` package). On the Browser cell side, the LwIP library was also already configured to be able to utilize this library. The only remaining work was to make our OS personality initialize the virtual Ethernet channel and set up the connection. The initialization function in okl4_basicio_rtx2 was modified to perform this initialization and pass the virtual Ethernet interface into LwIP. The virtual interrupt raised on receiving a message into this channel was also configured to call a `process_network_data` function implemented in okl4_basicio_rtx2, which simply passes the call on to the virtual Ethernet library.

To boot Android, the Android root filesystem is required to be present on an MMC/SD card inserted into the Beagle Board. The process of obtaining, building, and writing this root filesystem is described in the OKL4 SDK User Guide.

Upon Android boot, the physical network interface must be brought up, along with the bridged interface between the two cells. This required the `brctl` and `ifconfig` functions, neither of which is appropriately implemented in the Android shell. To fulfill this requirement, we compiled the `busybox` application, version 1.15.2 (with an appropriate patch to add brctl support), for Android and added it to the Android root filesystem on the SD card. We then wrote a script, `oknetup.sh`, to perform the network bring up, and modified the init process of Android to call this script.

Once this was completed we had an operational network driver cell. It should be noted that the provided Android has some dependencies on some other virtualized driver cells (such as vi2c); these are also included in our system configuration in order to support this network driver cell.

4.9 Miscellaneous Libraries

libpng (version 1.4.1), zlib (version 1.2.5), and libxml2 (version 2.7.7) were all ported to our programming environment. libpng and zlib were extremely straightforward. libxml2 required some more detailed configuration (to not build in http or ftp support), and required a single minor manual modification to not call the `getcwd` function, for which we have no implementation.

4.10 OWB BAL / WebKit

The Origyn Web Browser, version 1.0 (codenamed "Pukapuka") was ported to execute in our LWEE, using our developed OS personality and programming environment.

4.10.1 Loader Application

The included SDL loader application (WebKitTools/OWBLauncher/SDL/main.cpp) was modified to request a URL to be entered upon load. This served two purposes: it allows us to control the web page which is to be loaded, and it allows the Browser Cell to block while the Android network driver cell is fully booted and the virtual Ethernet interface configured.

4.10.2 Configuration and Build

The included CCMake build system was used to configure the initial environment for our build. All optional components in WebKit were disabled (such as WML, SVG, and Plugin support). The core components abstracted by the BAL were configured to use appropriate implementations, as follows:

- USE_FILESYSTEM_ACCESS: POSIX
- USE_FONTS: EMBEDDED
- USE_GRAPHICS: SDL
- USE_I18N: GENERIC
- USE_NETWORK: CURL
- USE_THREADS: PTHREADS
- USE_TIMER: LINUX

The "Generic" and "Embedded" options for Fonts and Internationalization, respectively, represent in-built implementations within OWB, specifically for embedded systems. As mentioned, this significantly eases the porting process for new embedded platforms.

The location of the SDL libraries, and various settings to define the toolchains to use and the compiler flags, were also configured. In particular, the location of the include directory that contains our set of LWEE OS personality and programming environment headers was defined.

Following this configuration, errors were still reported on building the Web Template Library (wtf) within JavaScriptCore. It appears that wtf requires its own configuration, which involved modifying a `Platform.h` header within wtf, defining an "OKL4" platform upon a compilation variable ("-Dokl4") being defined, and modifying various configuration options upon the definition of this platform. These definitions were as follows:

```
#define HAVE_STRINGS_H 0
#define HAVE_MMAP 0
#define HAVE_SYS_TIME_H 0
#define WTF_USE_PTHREADS 1
#define HAVE_PTHREAD_NP_H 1
#define HAVE_POSIX_MEMALIGN 1
#define HAVE_SBRK 1
#define HAVE_TM_GMTOFF 1
#define HAVE_TM_ZONE 1
#define HAVE_TIMEGM 1
```

Once these options were configured, a build of JavaScriptCore, WebCore, and WebKit proceeded smoothly, resulting in the libraries `libjsc`, `libwebcore`, `libwebkit-owb`, and `libwtf`, and finally the linked binary `owb`.

4.10.3 Browser Cell Definition and Merging

The OKL4 System Configuration file listed in Appendix A was constructed. Of particular note is the definition of the "browser cell":

```
<microvisor_cell name="browser" file="../Pukapuka/build_ok2/bin/owb"...> … </>
```

As can be seen, the browser cell uses as its binary file the compiled and linked "owb" binary from the OWB Pukapuka build. The browser cell also receives capabilities to use the resources of the timer, nand, and display devices that are defined in the board configuration file (which not listed for brevity). Capabilities are also given to the network and serial channels, allowing communication with the virtual Ethernet network device cell (Android) and the debugging serial device cell. Finally, options are configured in the cell environment for the network address and network gateway, allowing these to be modified if the system is booted on a different network configuration (it should be noted that the oknetup.sh script in Android should also be modified if the network topology changes).

Once this configuration was determined, it was passed to the `ok merge` command, which produces a system image that can be booted on the Beagle Board. Upon booting, once the network address to the machine hosting the simple HTML file is entered into the browser cell console, the HTML file is retrieved, parsed, rendered, and displayed on a connected display unit.

4.11 Miscellaneous Porting Issues

The following miscellaneous issues and challenges occurred during the porting process:

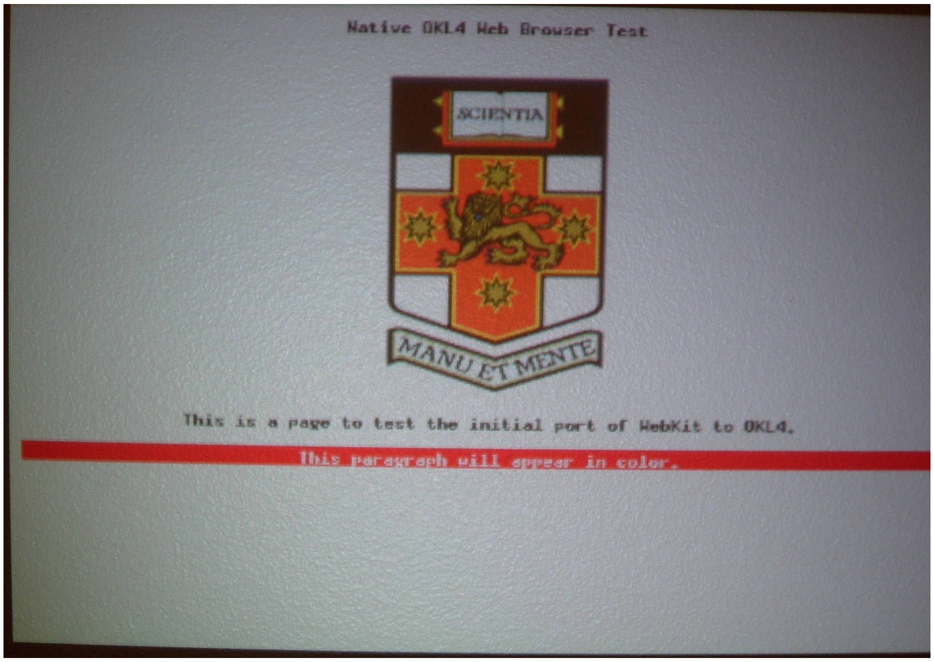
- The *arm-none-eabi* toolchain provides a basic libc and libstdc++ implementation, along with support for compiling C++ applications. However, we had to modify the default linker scripts provided with the OKL4 SDK in order to correctly link C++ applications in our environment (by correctly adding *exidx* sections to the linked executable).

- The above linker script modification, while resulting in correct C++ application linking, uncovered an existing limitation in the OKL4 SDK `ok merge` tool. Over 7000 sections are produced in certain segments of the owb binary; the ok merge tool section handling algorithm currently exhibits O(n^2) behavior. This resulted in the merge of the system configuration file taking between 7 to 10 minutes on a development machine. As a merge was required to be done after every change, prior to booting the board, this seriously hampered development progress.

Chapter 5

Evaluation

5.1 Rendering a Basic Web Page



Native OKL4 Web Browser Test

This is a page to test the initial port of WebKit to OKL4.

This paragraph will appear in color.

Figure 5.1: Photograph of native OKL4 render of basic test HTML page on Beagle Board (using the Pico DLP projector display)

A basic web page consisting of HTML, CSS, and an image was constructed as the canonical test of the success of the port. The listing of the web page appears in Appendix B. The page was hosted on a web server on a local area network, to which the Beagle Board was connected via a hub. Tests of loading the page on both the native OKL4 web browser image, along with a Linux-based Beagle Board system (Angstrom), were performed. The native OKL4 web browser renders the basic web page correctly, consistent with the native Linux system.

5.2 Performance Analysis

While performance was not a goal of the thesis, it is interesting to do an initial high-level comparison of the loading times of the basic web page on the native OKL4 web browser vs a Linux native implementation (Angstrom). The page was loaded, from a clean cache, 5 times, with the time from navigation to rendering measured using a stopwatch, and the results averaged.

System	Load time (avg)
OKL4	4s
Linux	2s

The OKL4 native browser exhibited significantly higher (50%) load times over Linux for the basic web page. As performance was not the goal of this thesis no further investigation was performed, however this area might form an interesting basis for future work. Certainly, theoretically, the majority of web browsing is network-bound and a native OKL4 web browser should exhibit performance on-par with alternative implementations, under the same network conditions.

5.3 Conclusions

We have successfully achieved the goals of the thesis: to port enough of the WebKit architecture to display a basic web page composed of HTML, images, and CSS.

Chapter 6

Future Work

6.1 Limitations

Our WebKit port currently suffers from the following limitations:

- *No head requests*: links to resources in the head of the html file (such as to an external CSS or JavaScript file) result in the download process stalling. We suspect it has something to do with Curl or LwIP threads performing multiple simultaneous requests and failing, but did not have time to investigate. It was not necessary to link to resources in the head of our example html files to achieve our goals, as, for example, the contents of the CSS or JavaScript could simply be copied to an equivalent <style> or <script> section, respectively.

- *No JavaScript execution*: although the dependencies for JavaScriptCore were implemented (since it was easier to do so than to remove libwtf from the JavaScriptCore package), JavaScriptCore itself currently fails on JavaScript execution. We did not have time to investigate, and it was not necessary to get JavaScript execution operating to achieve our goals.

- *Single page load*: the loader application only allows a single page to be loaded, with no support for navigation or additional page loads. This was not necessary to achieve our goals.

- *Performance*: the performance of the native OKL4 browser could certainly be improved. While it was not the goal of this thesis, it would be interesting to analyze and optimize performance where possible.

6.2 Porting Improvements

In addition to overcoming the above limitations, the following improvements to the port could potentially be made:

- *User interaction*: an implementation of, for example, keyboard and mouse support to enable user interaction would increase the functionality of the port. Alternatively, a separate navigation cell could be constructed which allows the user to type commands that get posted to the SDL event loop to control navigation from the conole.

- *Improved network support*: an implementation of a native OKL4 network driver, perhaps by use of a daughter board to the Beagle Board that adds an Ethernet port (or by use of a different board), would obviate the need for the additional Android network driver cell and associated cells.

- *Improved graphics support*: Open Source OpenGL drivers exist for the SGX graphics chip on the OMAP3530. It would be interesting to implement these drivers as native OKL4 drivers, which would enable the native OKL4 browser to use OpenGL as a graphics backend as opposed to SDL.

- *Improved WebKit support*: it would be interesting to enable additional functionality within WebKit for the native OKL4 browser, such as SVG or plugin support.

6.3 Secure Browser Architectures

Of course, the really interesting work will be to investigate the implementation of secure browser architectures on OKL4. There are at least two potential avenues of investigation:

- *Component Isolation*: there are a number of opportunities to exploit the isolation properties of OKL4 cell system architecture. For example, individual components of WebKit, such as the rendering engine or JavaScript core, could be isolated in separate cells to reduce the impact of attacks. Separate web applications could be loaded in individual cells, while still potentially sharing parts of the architecture. Security critical storage components, such as cookie jars offline storage, could be isolated in secure data storage cells. The WebKit2 architecture [9], work on which is just beginning, provides a split process model with an

IPC mechanism that would also be interesting to investigate implementing on OKL4. Finally, architectures like Gazelle [1], a secure web browser constructed as a multi-principal OS, would certainly be interesting to implement on OKL4.

- *Security policy implementation:* web browser security policies, such as the Same Origin Policy (SOP) [10], could potentially take advantage of OKL4 capabilities to guarantee that web applications only have access to those resources to which access has been granted.

Appendix A

System Configuration Listing

```
<?xml version="1.0" ?>
<!DOCTYPE image SYSTEM 'weaver-3.3.dtd'>
<image>
    <include file="boards/beagleboard-c3/beagleboard-c3.xml"/>
    <virtual_pool name="v-normal">
        <!-- Provide 1MB zero page region -->
        <memory base="0x00100000" size="0xf0000000"/>
    </virtual_pool>

    <!-- Physical memory allocation -->
    <physical_pool name="p-kernel"><memory base="0x80000000" size="8M"/></physical_pool>
    <physical_pool name="p-i2c"><memory base="0x80800000" size="512K"/></physical_pool>
    <physical_pool name="p-vaudio"><memory base="0x80880000" size="512K"/></physical_pool>
    <physical_pool name="p-serial"><memory base="0x80900000" size="512K"/></physical_pool>
    <physical_pool name="p-system"><memory base="0x80980000" size="512K"/></physical_pool>
    <physical_pool name="p-browser"><memory base="0x80A00000" size="60M"/></physical_pool>
    <physical_pool name="p-linux"><memory base="0x84600000" size="180M"/></physical_pool>

    <!-- OKL4 Kernel -->
    <kernel physpool="p-kernel" virtpool="v-normal">

        <!-- Physical Devices -->
        <use_device name="timer_dev"/>
        <use_device name="serial_dev"/>
        <use_device name="interrupt_dev"/>

        <!-- Linux timer needs these -->
        <use_device name="kprcm_dev"/>
        <use_device name="timer12_dev"/>
    </kernel>

    <!-- WebKit-based Browser Cell -->
     <microvisor_cell name="browser" file="../Pukapuka/build_ok2/bin/owb" kernel_heap="32K"
max_caps="32" physpool="p-browser" priority="2">
      <heap size="10M"/>
      <stack size="1M" />

      <environment>
        <entry key="serial_channel_cap" cap="/serial/cons_1"/>
          <entry key="network_channel_cap" cap="/browser/network" />
          <entry key="network_msg_size" value="2048" />
          <entry key="network_addr" value="0xC0A80296" /> <!--192.168.2.150 -->
          <entry key="network_mask" value="0xFFFFFF00" />
          <entry key="network_gateway" value="0xC0A80201" /> <!--192.168.2.1 -->
          <entry key="network_is_primary" value="1" />
      </environment>

      <!-- Serial -->
        <virq name="serial_channel_send_irq" source="serial/cons_1.in_not_full"/>
        <virq name="serial_channel_recv_irq" source="serial/cons_1.out_not_empty"/>

      <!-- Network -->
        <channel name="network" message_size="2048" max_messages="50" />
        <virq name="network_recv" source="browser/network.in_not_empty"/>

      <!-- Display -->
      <memsection name="framebuffer" size="0x200000" direct="true" cache_policy="writethrough"/>
      <use_device name="display_dev" />

      <!-- Timer -->
        <use_device key="timer_dev" name="timer2_dev"/>

      <!-- Nand -->
      <use_device name="nand_dev" />

    </microvisor_cell>

    <!-- Serial Server Cell -->
    <microvisor_cell file="applications/serial-server_beagleboard-c3" kernel_heap="32K"
name="serial" physpool="p-serial" priority="3">
      <heap size="4k"/>
        <environment><entry key="omap_serial_irq" value="74"/></environment>
        <channel name="cons_0" max_messages="8" message_size="32"/>
        <channel name="cons_1" max_messages="8" message_size="32"/>
        <channel name="cons_2" max_messages="8" message_size="32"/>
        <virq name="cons_0_in_not_empty" source="serial/cons_0.in_not_empty"/>
        <virq name="cons_1_in_not_empty" source="serial/cons_1.in_not_empty"/>
        <virq name="cons_2_in_not_empty" source="serial/cons_2.in_not_empty"/>
        <use_device name="serial_dev"/>
    </microvisor_cell>

    <!-- All below cells are required for network -->

    <microvisor_cell name="i2c_server" file="applications/i2c-server_beagleboard-c3"
kernel_heap="64k" physpool="p-i2c" priority="2">
        <heap size="64k" />

        <channel name="i2c1_0" max_messages="16" message_size="32" />
        <channel name="i2c1_1" max_messages="16" message_size="32"/>

        <environment>
            <entry cap="/system/ch0" key="syscell_client_i2c_kcap"/>
        </environment>

        <virq name="i2c1_0_in_not_empty" source="i2c_server/i2c1_0.in_not_empty"/>
        <virq name="i2c1_0_out_not_full" source="i2c_server/i2c1_0.out_not_full"/>

        <virq name="i2c1_1_in_not_empty" source="i2c_server/i2c1_1.in_not_empty"/>
        <virq name="i2c1_1_out_not_full" source="i2c_server/i2c1_1.out_not_full"/>

        <virq name="syscell_client_i2c" source="system/ch0.out_not_empty"/>

        <use_device name="i2c1_dev"/>
    </microvisor_cell>

    <microvisor_cell name="vaudio_server" file="applications/vaudio-server_beagleboard-c3"
kernel_heap="64k" physpool="p-vaudio" max_caps="40" priority="2">
        <heap size="64k"/>

        <!-- max_messages of 'ctrl' should be greater than that of 'data' -->
        <channel max_messages="9" message_size="32" name="ctrl0"/>
        <channel max_messages="8" message_size="4096" name="data0"/>
        <channel max_messages="9" message_size="32" name="ctrl1"/>
        <channel max_messages="8" message_size="4096" name="data1"/>

        <virq name="audio_0_ctrl_not_empty" source="vaudio_server/ctrl0.in_not_empty"/>
        <virq name="audio_0_ctrl_not_full" source="vaudio_server/ctrl0.out_not_full"/>
        <virq name="audio_0_data_not_empty" source="vaudio_server/data0.in_not_empty"/>
        <virq name="audio_0_data_not_full" source="vaudio_server/data0.out_not_full"/>
        <virq name="audio_1_ctrl_not_empty" source="vaudio_server/ctrl1.in_not_empty"/>
        <virq name="audio_1_ctrl_not_full" source="vaudio_server/ctrl1.out_not_full"/>
        <virq name="audio_1_data_not_empty" source="vaudio_server/data1.in_not_empty"/>
        <virq name="audio_1_data_not_full" source="vaudio_server/data1.out_not_full"/>

        <virq name="i2c_in_not_full" source="i2c_server/i2c1_1.in_not_full"/>
        <virq name="i2c_out_not_empty" source="i2c_server/i2c1_1.out_not_empty"/>

        <virq name="syscell_client_vaudio" source="system/ch1.out_not_empty"/>

        <virq name="vconsole-rx" source="serial/cons_2.out_not_empty"/>
        <virq name="vconsole-tx" source="serial/cons_2.in_not_full"/>

        <environment>
            <entry key="serial_channel_cap" cap="/serial/cons_2"/>
            <entry key="i2c_client_channel_kcap" cap="/i2c_server/i2c1_1/secondary"/>
            <entry key="omap_audio_irq" value="17"/>
            <entry key="syscell_client_vaudio_kcap" cap="/system/ch1"/>
        </environment>

        <use_device name="mcbsp2_dev" />
    </microvisor_cell>

    <microvisor_cell file="applications/system-server_beagleboard-c3" kernel_heap="0x10000"
name="system" physpool="p-system" priority="2">
        <heap size="0x10000"/>
        <channel name="ch0" max_messages="2" message_size="32"/>
        <channel name="ch1" max_messages="2" message_size="32"/>
        <channel name="ch2" max_messages="2" message_size="32"/>
        <channel name="ch3" max_messages="2" message_size="32"/>
        <virq name="ch0_in_not_empty" source="system/ch0.in_not_empty"/>
        <virq name="ch1_in_not_empty" source="system/ch1.in_not_empty"/>
        <virq name="ch2_in_not_empty" source="system/ch2.in_not_empty"/>
        <virq name="ch3_in_not_empty" source="system/ch3.in_not_empty"/>
        <use_device name="kprcm_dev"/>
        <use_device name="scm_dev"/>
        <use_device name="tap_dev"/>
    </microvisor_cell>

    <linux_cell name="linux" file="/root/microvisor/linux-oklabs-android-beagleboard-2.6.29-
ok2/vmlinux" kernel_heap="3M" physpool="p-linux" max_caps="100" priority="2">
        <atags>
            <cmdline value="androidboot.console=ttyV0 console=vcon0,115200n8 root=/dev/mmcblk0p2
init=/init rootwait"/>

            <!-- audio -->
            <channel name="data_channel" cap="/vaudio_server/data1"/>
            <channel name="ctrl_channel" cap="/vaudio_server/ctrl1"/>
            <virq name="ctrl_available_interrupt" source="vaudio_server/ctrl1.out_not_empty"/>
            <virq name="data_available_interrupt" source="vaudio_server/data1.in_not_full"/>

            <!-- console  -->
            <channel name="vconsole" cap="/serial/cons_0/secondary"/>
            <virq name="vconsole-rx" source="serial/cons_0.out_not_empty"/>
            <virq name="vconsole-tx" source="serial/cons_0.in_not_full"/>

            <!-- i2c -->
            <channel name="i2c1_channel" cap="/i2c_server/i2c1_0/secondary"/>
            <virq name="i2c1_interrupt" source="i2c_server/i2c1_0.out_not_empty"/>

            <!-- system -->
            <channel name="system_channel" cap="/system/ch2/secondary"/>

            <!-- veth -->
            <channel name="veth" cap="/browser/network/secondary"/>
            <virq source="browser/network.out_not_empty" name="veth-rx"/>
        </atags>

        <use_device key="vtimer_dev" name="vtimer0_dev"/>
        <use_device name="serial1_dev"/>
        <use_device name="serial2_dev"/>
        <use_device name="serial_dev"/>

        <use_device name="32ktimer_dev"/>
        <use_device name="wdtimer2_dev"/>
        <use_device name="gpio1_dev"/>
        <use_device name="gpio2_dev"/>
        <use_device name="gpio3_dev"/>
        <use_device name="gpio4_dev"/>
        <use_device name="gpio5_dev"/>
        <use_device name="gpio6_dev"/>
        <use_device name="usb_host_dev"/>
        <use_device name="i2c1_dev"/>
        <use_device name="i2c2_dev"/>
        <use_device name="i2c3_dev"/>
        <use_device name="mmc1_dev"/>
        <use_device name="mmc2_dev"/>
        <use_device name="mmc3_dev"/>
        <use_device name="sdma_dev"/>
        <use_device name="usb_otg_dev"/>
        <use_device name="mmu2_dev"/>
        <use_device name="kprcm_dev"/>
        <use_device name="scm_dev"/>
    </linux_cell>


</image>
```

Appendix B

Test HTML File Listing

```
<html>
  <head>
    <title>Native Browser Test File</title>

    <style type="text/css">
      body {font-size: 75%; color: black; text-align: center;}
      p.color { background: red; color: white; }
    </style>

  </head>

  <body>
    <h1>Native OKL4 Web Browser Test</h1>

    <img src="unsw-logo.jpg" alt="UNSW" />

    <p>
      This is a page to test the initial port of WebKit to OKL4.
    </p>

    <p class="color">
      This paragraph will appear in color.
    </p>

  </body>

</html>
```

References

[1] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The Multi-Principal OS Construction of the Gazelle Web Browser. MSR Technical Report MSR-TR-2009-16, http://research.microsoft.com/apps/pubs/default.aspx?id=79655, 2009.

[2] The WebKit Open Source Project, http://www.webkit.org, 2010.

[3] The Origyn Web Browser Project, http://www.sand-labs.org/owb, 2010.

[4] Simple Direct Media Layer library, http://www.libsdl.org/, 2010.

[5] The Beagle Board Development Platform, http://www.beagleboard.org, 2010.

[6] The Curl Library, http://curl.haxx.se/, 2010.

[7] The Lightweight IP Library, http://savannah.nongnu.org/projects/lwip/, 2010.

[8] CodeSourcery 2009-q1 Toolchain Release, http://www.codesourcery.com/sgpp/lite/arm/portal/release830, 2009.

[9] WebKit2 Architecture, http://trac.webkit.org/wiki/WebKit2, 2010.

[10] Michal Zalewski, Same Origin Policy, Browser Security Handbook (Part 2), http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy, 2010.