International Journal of Distributed and Cloud Computing
7 (2) December 2019, 19-26
http://www.publishingindia.com/ijdcc

Virtualization of the Legacy VxWorks Application onto the Linux Platform

Krithika L.[1*] and Vishalakshi Prabhu H.[2]
[1]M.Tech Student, Department of Computer Science and Engineering, R. V. College of Engineering, Bengaluru, Karnataka, India. Email: krithikal.scn18@rvce.edu.in
[2]Assistant Professor, Department of Computer Science and Engineering, R. V. College of Engineering, Bengaluru, Karnataka, India. Email: vishalaprabhu@gmail.com
*Corresponding Author

Abstract: With the expansion in the deliverables by the IT industries, the smart organizations are deploying virtualization in an increased manner which helps to reduce the capital expenses and make the business operations more active, agile, and affiliated with the organization's business requirements. While an application needs to be upgraded, the virtualization comes into picture which results in application to be hardware independent and share the same set of resources with other applications running on the same virtual platform. A detailed study is made on the architectural details of VxWorks RTOS, Linux operating system, and the migration architecture. Further, the detailed procedure for migrating a legacy VxWorks application to a 32-bit embedded Linux platform has been explained. Subsequently, the steps for porting it to the 64-bit Linux platform has been detailed. Observations show the major changes to be updated in the VxWorks application to work on the Linux OS. It can be said that the OS abstraction layer can be considered as the floating layer which needs a major part of the changes to be made when porting an application from one platform to another.

Keywords: Embedded linux, Hardware virtualization, Migration, Virtualization, VxWorks to linux, VxWorks RTOS.

I. Introduction

Virtualization always has different meanings for users with different kinds of applications. Virtualization is defined as creating the software version of any application above a hardware platform and/or the Operating systems (OS) which presents the independent virtualized OS environments as depicted by Fig. 1. The embedded systems' virtualization dates back to recent years.In fact, as the embedded systems are becoming more powerful, the ARM processors have also been implementing hardware virtualization recently. The major advantages of doing this are well-defined: the reduction in the complexity if only a few processing environments are needed for the applications computing service and the lessened system cost. As the system can be built on a lesser number of hardware elements, enhancement in the product reliability may also be achieved.

VxWorks is a real-time operating system that is made up of the collection software which governs the hardware resources, along with providing the precise timing services to the specified user. The important feature of this OS is to supervise the tasks efficiently and obtain the interrupts with ease. The software for this proprietary OS has been originally developed by Wind River systems.

In this era of virtualization, whenever there is a requirement for upgrading a hardware-dependent application with a proprietary operating system, the consideration of its virtual counterpart seems to be more promising. Further, choosing a general-purpose operating system that yields to the real-time requirements of the application rather than proprietary and the much costlier operating system will also benefit for the purpose.

In this regard, Embedded Linux can be the better choice, as it provides hard real-time functionalities as VxWorks RTOS with the less OS usage cost and all the networking API's required for the applications communication functionalities. The benefits of embedded Linux over the proprietary RTOS may contain a stabilized kernel; the ability for reading, modification, and distribution of the codebase; contains a number of suppliers for software, support, and development; no tokens or licensing costs.

Once the application has been virtualized onto the Linux environment, further the application can be ported from a 32-bit environment to a 64-bit environment with minimum modifications onto the codebase to utilize all the benefits of the current-day OS specifications.

Section II gives the details of the research work carried out recently in the field of hardware virtualization technology. Section III gives the architectural details of the VxWorks Real-time Operating System, Generic Linux platform, and the architecture considered for migrating the application from the VxWorks platform to Linux followed by Section IV which provides a detailed description of the methodology used for the virtualization procedure. Section V includes the observations fetched after the virtualization procedure. In the end, Section VI provides the conclusion of the study that has been carried out on the topic.

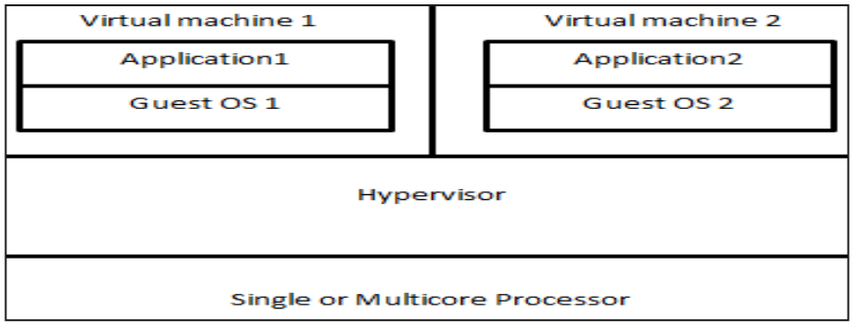

Fig. 1: Hardware Virtualization Platform

II. Literature Review

The topic of hardware virtualization is vast and applicable to various sizes of networks. A lot of research has been made on this topic and several researchers have written scientific articles and research papers on embedded virtualization.

The authors of the paper [1] give the details about the development of the hardware virtualization technology for systems with higher complexity. The paper provides the study of the framework developed for the virtualization concept for the hardware systems in the IT environment. The scenario of virtualization solutions has been proposed and the required methodology has been formulated for the hardware virtualization infrastructure.

In paper [2], the authors have proposed a framework and the programming model for the real-time applications that support access to the real-time CPU scheduling characteristics of the operating system. The main idea of the paper is to allow the applications to announce their temporal features and requirements for the CPU allocation. The authors say that the proposed framework in the research article will handle the set of heterogeneous requirements for the multi-core platform in order to explore the scheduling algorithms available in the kernel which match the requirements.

The embedded operating systems are being widely adopted in real-time applications where the task processing and real-time communication is required because of their strong and robust real-time performance. Nevertheless, the range of applications is restricted due to their unfriendly and costly user interfaces. As a result of this, there is an inclination towards the development of real-time applications on the common desktop operating systems and utilize their user-friendly GUI, multitasking capabilities, and exceptional hardware compatibilities. The paper [3] suggests a real-time extension for the Windows operating system for improving the scheduling compatibilities of the system. The authors also show that the proposed extension drastically improves the real-time performance of the operating system and allow the application to meet their required functionalities.

The authors of the paper [4] have proposed a real-time hardware VMM for real-time for the multi-core embedded systems which are called BlueVisor. In this research work, the design of the hypervisor and its implementation details are proposed. Further, it explains how the real-time requirements are met by exploiting a BlueVisor based virtualization environment with significant improvements in system performance and with the lessened performance cost.

In paper [5], the interrupt handling system used in a real-time virtualized environment, vINT has been proposed. This scheme runs a pseudo-VCPU construct that can be dedicated to the handling of interrupts. This construct will help in overcoming the limits that are imposed by the timing parameters of the virtual CPUs. vINT mechanism justifies and enforces the interrupt handling techniques and helps for execution flows in the guest virtual machine.

Few embedded processors will have the extensions to hardware virtualization and the hypervisors in this environment begin to use these extensions. These existing hypervisors will be usually designed for the general-purpose systems and will always rely on the larger stacks. Due to this, they may cause security problems and much more over-head in execution. To overcome this problem, a thin full-virtualization Type-1 VMM is proposed [6] which is used for the real-time embedded systems that use ARM processors.

The paper [7] presents a virtual CPU (vCPU) migration mechanism that improves the real-time responsiveness of the General-Purpose Operating System (GPOS). This mechanism is proposed for the multicore embedded hardware virtual environment as well as can be applied to the CPS environment. The virtualized embedded systems combine the applications with mixed-criticality on a multi-core platform. These systems will require higher performance solutions for the safer working of I/O systems. The proposed vCPU migration mechanism helps to achieve these performance improvements. In the paper [8], a hardware-based input/output virtualization mechanism that uses a memory management unit (MMU), memory-mapped I/O as well as I/O memory management unit (IOMMU). The author of the paper here compares the capacity of the PCI Express (PCIe) Single Root I/O Virtualization (SR-IOV) and the serial rapid I/O standards.

Recently the real-time functionalities of the Linux kernel are enhanced by developing the Linux RT patch. Paper [9] helps in evaluating the real-time features of the RT patch installed in the Linux kernel. This also helps the designer to understand the expectations of the real-time performance and further in deciding whether it is acceptable for the given group of

applications. A survey of methodologies had been presented in paper [10] that are been used by the commercial vendors and the open-source functional groups for enhancing the Linux kernel's real-time performance. The paper figures out the major factors that influence the real-time performance of the kernel and defines the approaches to improve the performance of the Linux kernel in real-time.

The white paper [11] describes the procedure used for mapping the legacy VxWorks based architecture onto the Linux, the other choices for the execution of the migrated application, the API and IPC translation of VxWorks, the enhancement in the reliability caused by the migration, the process of migration and the challenges and solution related to the specific application migration.

III. Architectural Details of the Operating Systems

This section gives the architectural details of the VxWorks Operating system, Linux Operating system as well as the architecture for migration from VxWorks to Linux OS.

VxWorks

VxWorks is the most widely known real-time OS. VxWorks acts as a better choice for developing embedded applications, as it provides a runtime environment that is reliable. VxWorks has more than 1800 APIs that make it more flexible and also easier to obtain the required functionality.

The VxWorks RTOS contains the kernel, many support packages, and the middleware for the device drivers as shown in Fig. 2 [12]. It also provides support for useful application support like the network stack, inter-process communication, and the file system. The applications, middleware, and other packages are separated from the kernel, which helps in easier bug fixing and for the testing of the new characteristics.

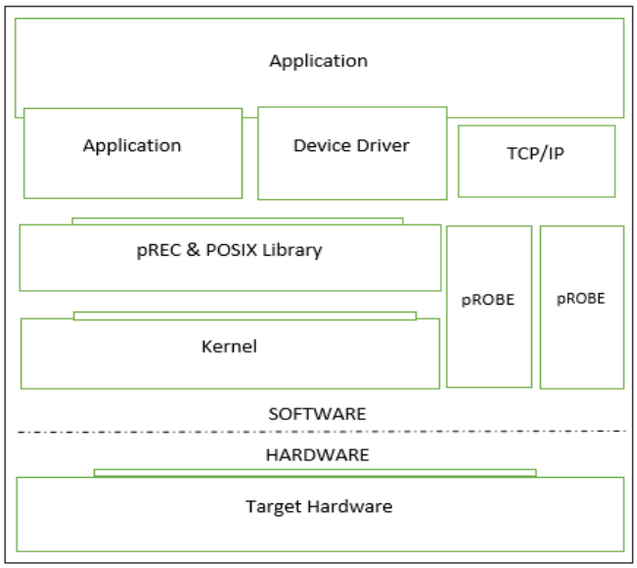

Fig. 2: VxWorks Architecture

Linux

Linux is the general-purpose operating system that was mainly developed to be used in the personal computers, but as it is been ported to many of the other platforms, Linux is now popular in its functionality as the server OS as well. The major difference between Linux and other operating systems is that the kernel is free and open-source software. Most of the Linux distributions allow a set of libraries to downloaded and installed with the available network connection. This helps the users to obtain their required development environment.

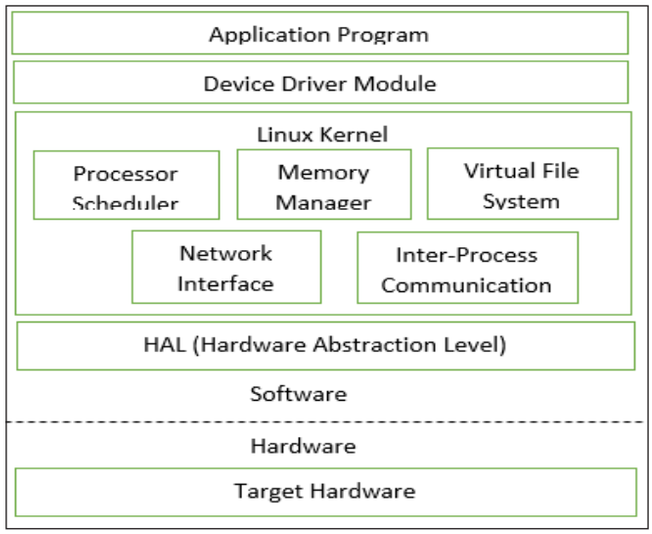

Fig. 3: Generic Linux Architecture

As shown in Fig. 3, Linux is having a layering structure. It is comprised of the Linux kernel, a Hardware Abstraction Layer (HAL), and many other functional modules [13]. A wide range of processors is supported by the hardware-dependent code provided in the HAL. The kernel is comprised of five subsystems: the scheduler, the network interface, the memory manager, the inter-process communication, and the virtual file system.As the kernel is monolithic, there is only one layer in the Linux Kernel. If any one of the subsystems is being changed, then the whole of the kernel should be built up again. For this modified kernel, device drivers can be dynamically attached.

Migration Architecture

Embedded virtualization involves the CPU partitioning, memory, and many other resources to accommodate the VxWorks and multiple guest application operating systems, usually the Linux OS, to host a higher-level software. Virtualization helps in migration by letting the VxWorks APIs as well as the Linux APIs to co-exists in the OS Abstraction layer which allows making minimal changes to the codebase, while the Linux will run in its other applications in parallel. The migration architecture as shown in Fig. 4, is helpful when the application has dependencies on the legacy VxWorks API's and its performance features, ex: real-time characteristics as well as any specific implementation of the protocol stacks.

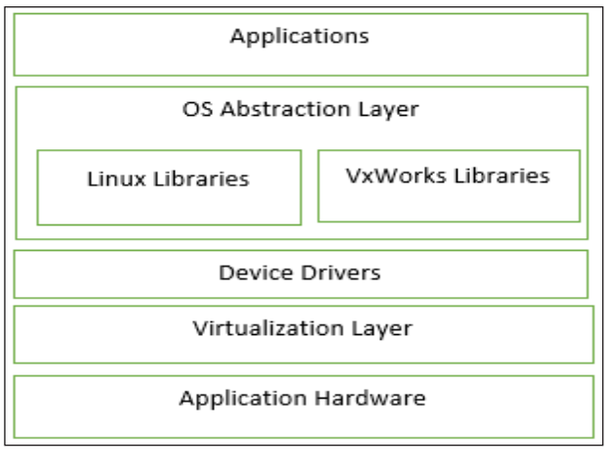

Fig. 4: Migration Architecture

IV. Methodology

While considering the migration of an application from VxWorks RTOS to embedded Linux, initially the application is ported to 32-bit Linux platform, as most of the legacy applications would work on 32-bit architecture and subsequently it is migrated to a 64-bit platform, so that it can be compatible with the present-day specifications of Linux. This section initially gives the methodology for migrating the VxWorks codebase to a 32-bit Linux platform and further onto 64-bit architecture.

A. Migration of the Application to 32-Bit Linux Platform

The steps involved in porting the VxWorks code onto the 32-bit Linux platform are as follows:

Step 1: Deciding on the Linux cross development toolset.

The first step in porting of the application from VxWorks to Linux include the Linux environment to be set up with all the required system software installed. The development toolset consists of the GNU C and C++ compiler, assembler, and the linker for the chosen target environment.

Step 2: Setting up the Linux environment on the target hardware.

As the Linux development toolset is decided and installed, the kernel and the subset of the required daemons and the other system utilities are ported onto the chosen hardware environment. The required set of device drivers may also be spawned. Many times, the drivers may already be available and only the recompilation may be enough, but in rare cases, new drivers must be generated.

Step 3: Recompilation of the VxWorks codebase on the new chosen environment.

Once the toolset is generated and the target environment with Linux kernel is set up, the next step would be eliminating the compiler errors in the application codebase on the target environment. This includes solving the differences in the contents of the header-files, their location, and also the function prototypes. As the VxWorks RTOS also implements many of the POSIX, UNIX, as well as the standard C runtime function calls in its built-in ecosystem, and some of these APIs may conflict with that of the Linux APIs. The solution for this is to find out those conflicting APIs and replacing them with that of the Linux counterpart headerfiles containing those APIs.

Step 4: Implementing the OS Abstraction layer.

Once the entire codebase of the VxWorks is successfully compiled on the established virtual environment with the Linux platform, the build process of the code may fail as there may be unresolved linker references. These references remain unresolved due to the functions that were present in the legacy VxWorks RTOS environment that may not be implemented again or duplicated on the Linux virtual environment. Some of the possible ways for eliminating these unlinked references are:

- Extending the virtualized target environment by developing the undefined VxWorks APIs.
- Substituting the existing Linux APIs those which provides similar functionalities as that of the VxWorks function calls,
- Defining the abstraction layer where both the APIs of the Linux as well as VxWorks co-exists. This becomes helpful when the application has a dependency on the VxWorks and to make only a few changes to the code base, or
- Re-developing the entire code base of the application in order to remove the requirement of the missing VxWorks API call.

For those of the VxWorks calls which are used for the initialization and manipulation of the devices, filesystems, network stacks, the application is re-implemented to remove the call by executing the similar functionality in Linux or by using the equivalent Linux functions instead.

Step 5: Debugging of the ported 32-bit application code.

The best method for debugging any source code is to use the tried and tested method of including the printf() trace statements while using the GDB tools. These statements will help in reporting the behavior of the codebase at the critical intervals. The included printf() statements can be controlled by the conditional compilation or via diagnostic option settings. Some discretion is also advised while using these trace statements, as they incur preferably amount of overhead while executing the code and may cause a reduction in the real-time performance, but preferably intolerable levels.

B. Migration of the Application to 64-Bit Linux Platform

Once the VxWorks code had been ported to the 32-bit Linux platform, further it must be migrated to a 64-bit platform. Some of the key differences between the 32-bit as well as 64-bit Linux platform [14] are:

- In 64-bit processors, the size of the registers and the width of the memory addresses is higher compared to the 32-bit platform.

- 64-bit processors have a greater number of integer and floating-point registers.
- A higher amount of Random-Access Memory (RAM) is supported by 64-bit Operating systems when compared to 32-bit OSes.

These differences should be eliminated while migrating the code from the 32-bit Linux platform to a 64-bit environment. The major issues to be taken care while migrating the application from 32-bit to 64-bit are as given below:

i. Variable Type Casting

While typecasting of the variable, there may be chances of the data loss due to the differences in the size of the datatypes. This may lead to incorrect behavior of the application due to incorrect data input. Ex: Typecasting of the string from char * to int will lose precision because the size of the char * and size of int is different in 32-bit and 64-bit machine.

- on 64-bit sizeof(char*) is 8 and sizeof(int) is 4
- on 32-bit sizeof(char*) is 4 and sizeof(int) is 4

To overcome this typecasting issue, the simple solution is to remove the typecasting in the 64-bit architecture.

ii. Arithmetic Operations on Pointer Addresses

```
unsigned int a, b, c;
int *p;
```

The above example works fine with the 32-bit platform if the value of "a*b*c" does not exceed the maximum value i.e., 4GB. But in the 64-bit platform, this value may get extended beyond the maximum specified value. While performing the multiplication operation the overflow will occur and the result will be converted to ptrdiff_t type. This further results in incorrect pointer calculations.

The solution for this problem is to use the explicit type conversions or the "memsize" types while working with pointer arithmetic operations.

iii. Memsize Types in Unions

The uniqueness of unions is that all the members of the union are allocated with the same memory i.e., they overlap each other. The unions with the pointers and the other members should be paid attention to. Considering the example:

```
union PtrNum {
char * ptr;
unsigned num;
} u;
```

The above code works on the 32-bit systems but not in the 64-bit platform. Here, while working with the member "num", we also work with the part of the pointer member "ptr". Hence, a datatype that relates to the pointer size must be used. The above code can be fixed as:

```
union PtrNum {
char * ptr;
size_t num; //fixing the datatype
```

iv. Operations of Bit Shifting

While porting the application from 32-bit platform 64-bit, bit shifts may cause a lot of problems, if it is not fixed properly. When considering the following example:

```
ptrdiff_t Bitset ( ptrdiff_t val, unsigned num ) {
ptrdiff_t mask = 1 << num;
```

The above code executes well in the 32-bit platform while defining the bits ranging from 0 to 31. When this code is migrated to a 64-bit platform, there will be a need to define the bits from 0 to 63. In the above code, the variable 1 is of int type and there will be an overflow while performing bit shift operations. This overflow can be overcome by typecasting variable 1 with the same types as that of the "mask" variable i.e., ptrdiff_t. So now the code will be fixed as:

```
ptrdiff_t Bitset ( ptrdiff_t val, unsigned num) {
ptrdiff_t mask = (ptrdiff_t) 1 << num;
```

Many of the other situations which are not discussed above may arise while porting an application from 32-bit to 64-bit architecture. Most of them can be tackled by making the use of "memsize" datatype rather than specific pointer datatypes.

V. Observations

This section describes the resulting model of the virtualized application for the specific requirements from the migration process. When an application running on the VxWorks operating system is virtualized onto the Linux platform the following gets updated with respect to the new platform.

A. Tasks in VxWorks are Mapped onto Threads

A task in VxWorks is a runnable unit. A task has a Task Control Block (TCB) which contains information regarding the task space, priority, etc. These tasks in VxWorks can be mapped onto processes or threads in the Linux operating system.

Processes are heavier when compared to threads in general as they usually carry more context. The Linux thread context contains a program counter, stack, subset of the CPU registers, but the process along with the above-mentioned information adds an entire virtual address space for the application context. The major effect of the process's heavyweight context is the process creation time and the inter-process context switching time. These effects of the process lead a task to be mapped onto a thread rather than the process itself. The mapping of a task in VxWorks onto Linux Threads is shown in Fig. 5.

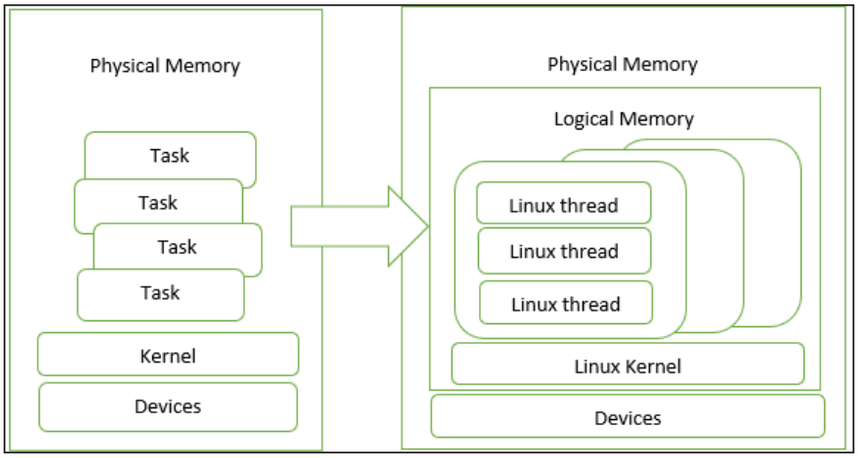

Fig. 5: Mapping Between VxWorks Tasks and the Linux Threads

B. Makefiles of the Application are Updated as per the Linux Standards With Necessary Libraries

Though VxWorks and Linux are POSIX standard operating system, the libraries implementation are different for each of them. The makefiles are files including the compiler information required by any application to be executed. The makefiles of the application containing the information regarding various libraries included and the compiler flags, that is running on the VxWorks platform are updated with the Linux standard libraries and Linux compliant flags as per the requirement. Fig. 6(a) shows the makefiles of the application running on the VxWorks operating system. Further, Fig. 6(b) gives the details included in the makefile as per Linux standards.

```
## build-configuration info
CPU              = PPC603
TOOL             = gnu
TOOL_FAMILY      = gnu
DEFAULT_RULE     = vxWorks
OPT              =


#include $(TGT_DIR)/h/make/defs.project


## build-configuration info
AR                  = arppc
AS                  = ccppc
CC                  = ccppc
CC_ARCH_SPEC        = -mcpu=603 -mstrict-align -mno-implicit-fp
CPP                 = ccppc -E -P
EXTRA_MODULES       = $(SUB_TARGETS)
LD                  = ldppc
LDFLAGS             = -X -N
LD_LINK_PATH        = -L$(TGT_DIR)/lib/ppc/PPC603/gnu -L$(TGT_DIR)/lib/ppc/PPC603/
LD_PARTIAL          = ccppc -r -nostdlib -Wl,-X
LD_PARTIAL_FLAGS    = -X -r
LIBS                = $(VX_OS_LIBS)
NM                  = nmppc
OPTION_DEFINE_MACRO = -D
OPTION_DEPEND       = -M -w
OPTION_GENERATE_DEPENDENCY_FILE = -MD
OPTION_INCLUDE_DIR  = -I
OPTION_LANG_C       = -xc
OPTION_UNDEFINE_MACRO = -U
SIZE                = sizeppc
TOOL_FAMILY         = gnu
POST_BUILD_RULE     =
POST_HEX_BUILD_RULE =


DEFINE_FLAGS +=   -DCPU=$(CPU) -DRW_DONT_USE_MEMORY_POOL -DVXWORKS \
          -DSYSTMR_IS_CPMTMR1 -DNDEBUG
```

Fig. 6(a): Makefiles With the VxWorks Libraries and Flags

```
# Objects are stored in core dependent directory, allows for simultaneous existance of debug and release objects
ifeq ($(strip $(OBJ_DIR)),)
OBJ_DIR = objs(CORE_NUMBER)
endif

## build-configuration info
AR              =
AS              =
CC              = g++
CC_ARCH_SPEC    =
CPP             =
EXTRA_MODULES   =
LD              =
LDFLAGS         =
LD_LINK_PATH    =
LD_PARTIAL      =
LD_PARTIAL_FLAGS =
LIBS            = -lpthread -lrt -lsnmp
NM              =

DEFINE_FLAGS +=

#Flags that have to do with CPU go here
COMP_ARCH_FLAGS =

COMP_MISC_FLAGS = -Wall -fpermissive -Wno-unused-but-set-variable

COMPILER_FLAGS = $(COMP_ARCH_FLAGS) $(COMP_MISC_FLAGS)

# ****************************************************************************
# Note: Template Instantiation uses the CFLAGS macro.
# Besides, we're using the C++ compiler for C files anyway...

CFLAGS  = $(DEFINE_FLAGS) $(COMPILER_FLAGS)  ${IFLAGS}
```

Fig. 6(b): Makefiles With the Linux Libraries and Flags

C. Implementation of the OS Abstraction Layer

The OS abstraction layer includes the VxWorks Wrapper class. The wrapper module includes all the VxWorks APIs that are present in the existing application codebase. These APIs are reimplemented with Linux APIs for providing specific functionalities. Some of the VxWorks library modules that need to be implemented in the abstraction layer are listed in Fig. 7.

Fig. 7: Implemented VxWorks APIs in the Abstraction Layer

The OS Abstraction layer can be considered as the floating layer of the proposed architecture. This layer is the only part where we need to make more of the changes during the porting process. The abstraction layer is the one that needs to be updated when porting an application from any of the platforms to others. This also includes the seamless integration of the APIs of both platforms.

VI. Conclusion

Virtualization generally means creating the abstraction of a hardware device or an application or an operating system and running it on the isolated virtual environment. A detailed study on the VxWorks RTOS, Embedded Linux, and architectures are carried out.

The architecture for migrating an application from VxWorks RTOS to Embedded Linux has been designed. A brief study is made on the methodology that is used in porting a legacy application from VxWorks RTOS to Embedded Linux. Initially, the steps used for migration from VxWorks to the 32-bit Linux platform have been discussed. In the later stage, the further porting of the application from a 32-bit Linux platform to a 64-bit environment has been presented.

It can be observed that the tasks of the VxWorks application are mapped onto the Linux threads rather than the processes as processes are heavier than the threads. It includes the updating of the makefiles of the application as per Linux standards. Further, the wrapper module is implemented in the Abstraction layer which includes the implementation of the important VxWorks APIs as per the Linux standards. This Abstraction layer implementation helps in making only a few changes in the existing application codebase and further helps in retaining the required functionality.

In the future, the application can be ported onto the upcoming architectures of the Linux platform by considering the pointer manipulation techniques as per the required architectures. As the application is virtualized to be hardware independent, it can be deployed on the cloud infrastructure to further minimizing resource usage costs.

References

[1] A. Erulanova, G. Yessenbekova, K. Zhanysbayeva, A. Tlebaldinova, Z. Zhantassova, and G. Zhomartkyzy, "Hardware and software support of technological processes virtualization," *7th International Conference on Electrical and Electronics Engineering (ICEEE),* Antalya, Turkey, 14-16 April 2020.

[2] G. Serra, G. Ara, P. Fara, and T. Cucinotta, "An architecture for declarative real-time scheduling on linux," *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC),* Nashville, TN, USA, 19-21 May 2020.

[3] K. Chen, C. Du, J. Chen, and Z. Yuan, "Real-time extension technologies on the windows operation system," *2019 IEEE 8th Joint International Information Technology and Artificial Intelligence Conference (ITAIC)*, Chongqing, China, 24-26 May 2019.

[4] Z. Jiang, N. C. Audsley, and P. Dong, "BlueVisor: A scalable real-time hardware hypervisor for many-core embedded systems," *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),* Porto, Portugal, 11-13 April 2018.

[5] H. Li, M. Xu, C. Li, C. Lu, C. Gill, L. Phan, I. Lee, and O. Sokolsky, "Multi-mode virtualization for soft real-time systems," 2018 *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),* Porto, Portugal, 11-13 April 2018.

[6] H. Kim, S. Wang, and R. Rajkumar, "Responsive and enforced interrupt handling for real-time system virtualization," *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,* 2015.

[7] T.-H. Lin, H. Mitake, and T. Nakajima, "Improving GPOS real-time responsiveness using vCPU migration in an embedded multicore virtualization platform," *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications,* Taipei, Taiwan, 19-21 August 2013.

[8] D. Munch, O. Isfort, K. Muller, M. Paulitsch, and A. Herkersdorf, "Hardware-Based I/O virtualization for mixed criticality real-time systems using PCIe SR-IOV," *2013 IEEE 16th International Conference on Computational Science and Engineering*, Sydney, NSW, Australia, 3-5 December 2013.

[9] W. Betz, M. Cereia, and I. C. Bertolotti, "Experimental evaluation of the Linux RT patch for real-time applications," *2009 IEEE Conference of Emerging Technologies and Factory Automation*, Mallorca, Spain, 22-25 September 2009.

[10] N. Vun, H. F. Hor, and J. W. Chao, "Real-time enhancements for embedded linux," *2008 14th IEEE International Conference of Parallel and Distributed Systems,* Melbourne, VIC, Australia, 8-10 December 2008.

[11] B. Weinberg, "Migrating legacy VxWorks applications to linux," MontaVista Software, version 3.0.1, September 2008.

[12] Wind River VxWorks Platforms 6.9 documentation. [Online]. Available: http://www.windriver.com

[13] G. Newell, "Analysis of the top 10 linux operating systems," [Online]. Available: http://www.everydaylinuxuser.com/2014/02/analysis-of-top-10-linux-operating.html

[14] D. Pavan, and S. Gaonkar, "Porting 32-bit software to 64-bit platforms," White Paper by Ittiam Systems Pvt Ltd., 2015.