Eliminating the Operating System via the Bare Machine Computing Paradigm

Uzo Okafor, Ramesh K. Karne, Alexander L. Wijesinha, and Patrick Appiah-Kubi
Department of Computer and Information Sciences
Towson University
Towson, MD 21252 USA
uokafo1@students.towson.edu, {rkarne, awijesinha, appiahkubi}@towson.edu

Abstract - **Computer applications typically run under the control of intermediary system software that is in the form of an operating system such as Windows or Linux, or a small kernel. The application could also be embedded within the operating system or kernel itself. This paradigm makes applications dependent on an intermediary software layer. An alternative approach is to eliminate this layer by writing computer applications that can run directly on the hardware. This approach takes a small or tiny kernel to its extreme, eliminating the operating system, which results in a novel bare machine computing paradigm. In this paper, we describe the bare machine paradigm, and illustrate how to build self-supporting bare machine applications by eliminating application dependence on an operating system or kernel. The new paradigm requires that the developer be aware of the underlying hardware resources and use them efficiently for the needs of a given application suite. We also describe a set of generic bare interfaces that can be used across many pervasive devices as well as ordinary desktops and laptops. These interfaces have made it possible to build large bare applications. The bare machine paradigm paves the way for software interfaces to be incorporated into a chip, introducing a computing model where applications are independent of any intermediary software.**

Keywords - bare machine applications; bare machine computing; middleware; direct hardware interfaces; operating systems.

I. INTRODUCTION

Building bare machine applications, which are independent of any intermediary software, is daunting due to constraints imposed by the existing computer architecture and development environments. Most CPUs are designed to work with an operating system (OS) or kernel and do not provide any interfaces to directly control the hardware. In some cases, the kernel or virtual machine may allow an application direct hardware access, but does not fully relinquish its control to the application. However, for certain specialized applications and secure systems, even the presence of a small kernel may prevent the application from fully controlling its environment and managing the hardware.

We propose to eliminate the OS (or kernel) and give full control to applications. These applications are then able to run on the bare hardware without the need for any additional software layers. There is no persistent storage or any other resource to secure on a bare machine, device, or computing system. Moreover, only one bare application suite runs at a time. When an application is not running, the machine is not running any other code. It simply has memory, processors and an I/O controller to communicate with the applications when needed. Instead of an OS or kernel providing resources, an application suite manages the hardware. This does not mean that the applications replicate OS functionality. Rather, applications only contain code that is required for a given application suite. An application suite is modeled as an Application Object (AO) [6] that carries its own application and execution environment. For example, an AO may consist of a text processor/editor, a Webmail client, and a Web browser, and bare interfaces to the hardware. An AO programmer needs to have knowledge of the underlying resources, since an AO controls and manages all the hardware when it runs. A bare machine user carries a removable mass storage device to boot, load and run the application suite, thus making the machine bare when the AO is not loaded (since no OS or kernel is needed to run the suite).

When such bare machines are built, they become ownerless and can be used by anyone, anytime, and anywhere. Many complex bare applications have been built to illustrate the bare machine computing (BMC) paradigm. These include a Web server [4], Webmail server [1], conventional (non-HTTP) email server and client, VoIP soft-phone [9], SIP server, and bare PC clusters using split servers [13]. The development of such applications served as the motivation for designing the direct hardware interfaces to a bare PC (x86 architecture). These interfaces are generic and can be used to construct any bare machine application. One can make these interfaces and the BIOS part of the hardware in the future, thus creating a pure BMC environment, where there is no other software needed to run computer applications. A high-level methodology for developing bare machine applications was outlined previously [10]. Here, we provide details of how to develop such bare machine applications by using a set of generic hardware interfaces. In particular, this paper describes the direct hardware interfaces needed to eliminate the intermediary OS.

The rest of the paper is organized as follows. Section II provides the motivation for this work. Section III describes the bare machine computing paradigm and its characteristics. Section IV illustrates the development of bare machine applications using a step-by-step process. Section V presents the direct generic hardware interfaces. Sections VI, VII and VIII respectively cover the use of a bootable USB, memory map, and the novel features of this approach. Finally, Section IX gives the conclusion.

 ISBN: 978-1-61208-272-1

II. MOTIVATION

The following considerations serve as the motivation for developing BMC applications based on the underlying paradigm: the proliferation of operating systems (OS) and frequent new releases to replace them; the rapid obsolescence of existing computer applications; the myriad of programming languages and interfaces; and the heterogeneity of computer architectures and platforms. While many arguments can be given to support the current evolution of conventional systems/platforms and their advantages, the BMC paradigm has been shown to be feasible for building a variety of complex applications such as the servers, clients, and high-performance systems noted above.

Most current systems use complex concurrency control mechanisms, paging, virtual memory, and other well-known concepts that have evolved due to lack of large memory, memory costs, shared resources, and the need to serve multiple users. Under those conditions, computing evolved towards complex systems with rapid obsolescence and less security. The BMC paradigm [7] enables applications to directly communicate with hardware, thus eliminating all middleware. Using this paradigm, applications can be written in C/C++ or other languages, where an AO programmer can directly call hardware interfaces, as originally proposed in [8]. This paper extends those ideas to address general purpose development of bare machine computing applications and a set of generic interfaces to the hardware.

III. BARE MACHINE COMPUTING PARADIGM

The BMC paradigm was originally referred to as dispersed operating system computing (DOSC) [7], but we have seen further evolution of DOSC into the BMC concepts as shown in Fig 1. A conventional OS, kernel or embedded software acts as middleware between the hardware and an application. An application programmer is isolated from an application's execution environment, resource control and management.

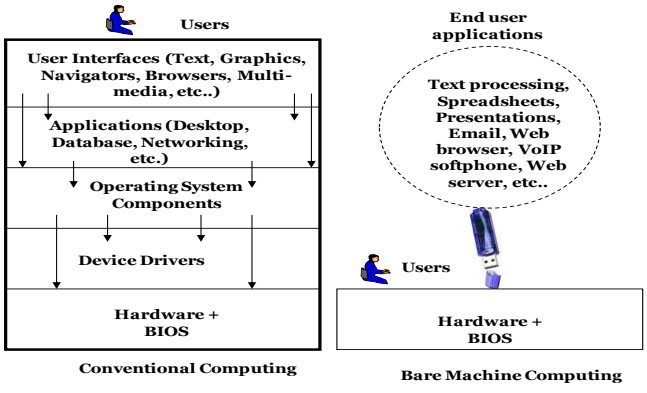

Figure 1. Conventional OS versus BMC paradigms

That is, the programmer has no direct control of the program's execution or the resources needed. In the BMC paradigm as shown in Fig 1, the OS is eliminated and the AO programmer is totally responsible for managing hardware resources. The AO programmer has knowledge and full control over a given application as well as its execution. Each AO only carries its needed controls and the direct hardware interfaces. The AO programmer is a domain expert for a given set of applications that are contained in a given AO. The BMC paradigm differs from conventional computing in two major ways. First, the machine is bare with no existing software and protected resources. Second, an AO programmer controls the program's execution and manages the hardware.

The BMC paradigm makes a computing device owner-less and simplifies the design of secure systems since there are less avenues of attack and no underlying middleware that an attacker can control. Viewed another way, when a device is bare and contains no valuable resources such as a hard disk or kernel, there is nothing to own or protect. In BMC, mass storage is external and detachable. The mass storage can also be on a network. In this approach, an AO is built for a given set of applications to run at a time on a machine as a single monolithic executable. The boot, load, executable, data and files are stored on a mass storage device such as a USB. When a USB is plugged into a computer, the machine boots and runs its own program without using any extra software or external programs. This implies that no dynamic link libraries (DLLs) or virtual machine code are allowed in this approach. What runs in the machine, is exactly what has been loaded (and nothing else).

This computing paradigm is different from conventional computing approaches since it is based on applications instead of computing environments. This is not a mini-OS or kernel, as there is no centralized program running in the machine to manage resources. Instead, the resources are managed by the applications themselves and run without using any OS/kernel or intermediary software.

A variety of attempts have been made to eliminate OS abstractions or bypass the OS. However, none eliminate the kernel or OS altogether. Thus, while the BMC paradigm resembles approaches that reduce OS overhead and/or use lean kernels such as Exokernel [2], IO-Lite [12], Palacio [11], and the Hardware Abstraction Layer [14] in Java, there are significant differences. These include self controlled applications and programmer-driven execution, and the lack of centralized code that manages system resources. A model to analyze tradeoffs between feature-rich and minimalist or "barebone" systems is presented in [15]. While such minimalist systems usually require an operating system or kernel, they may have also some characteristics in common with BMC systems.

IV. BMC APPLICATION DEVELOPMENT

In BMC, a suite of applications such as a text processor, Webmail server and Web browser can be bundled together and run without any OS or kernel support. Fig. 2 illustrates the major steps involved in developing

BMC applications. First, a choice has to be made about the suite of applications; next, the architecture of the CPU on which they will run has to be identified. Using today's CPUs, constructing a BMC application is a daunting task as they provide neither direct hardware interfaces, nor a development environment that facilitates building applications independent of an OS. For example, a bare PC requires the BIOS to boot, and an ARM processor requires a UBOOT tool. The program counter of a given processor is not directly accessible to the programmer. In a machine with an x86 CPU, the program counter can only be loaded by jumping to the task segment, where its value is stored and updated by the CPU. In a bare machine application, the program counter must be handled inside an application and not controlled by an OS or other software.

Memory needs or requirements must be considered for a given application's code, data and stack. The application programmer has to determine memory areas for the code, data and stack, as these applications run in a real memory. Real memory is cheap and affordable today. It is therefore feasible to avoid paging and virtual memory overhead, and the associated management. The absence of any other software in the system eliminates many unnecessary features commonly found in today's technology. Most BMC application suites only require small amounts of memory compared to OS-based applications. For very large applications, one can use mass storage to provide extended storage using swapping techniques. Section 7 describes details of the memory map created for some real-world applications using the BMC paradigm.

The next step is to construct an application suite using programs that are independent of any OS. This application suite should be able to run on any compatible CPU without changes or adaptations. Different CPU architectures have different compilers to compile code. This requires that I/O related code be identified and direct hardware interfaces be deployed. One of the key elements in writing BMC code is being able to differentiate between code that is OS dependent, code that is OS independent and code that is I/O related. For example, file I/O is OS dependent code and a for-loop is OS independent code. User interfaces to support keyboard, mouse and display are all I/O related code.

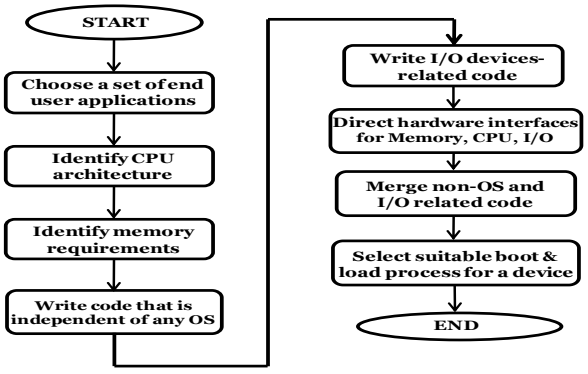

Figure 2. Steps in developing bare machine applications

Once OS dependent code and I/O related code are written (as hardware interfaces), they can all be integrated with the rest of the OS independent code and run as a single monolithic executable. The above approach introduces many challenges that must be addressed when developing BMC applications. They include the boot-up process and loading of an application suite. Each computing device is different in its boot process and the internal details are often hidden. Similarly, loading an application on a bare device also poses difficulties as it requires readily available tools that are OS dependent. Developing an OS independent loader requires a thorough knowledge of the CPU architecture and its development environment. Domain knowledge and related expertise for each CPU device are required to develop the bare boot and load processes.

V. DIRECT HARDWARE INTERFACES

Conventional computer applications and programming languages use OS calls or system calls injected at link time from an OS such as Windows [16] or Linux [3]. These calls include memory, keyboard, terminal screen, network, mass storage, and interrupts. Some OSs include in their repertoire other commonly used OS-independent functions such as memory copy, string operations and concurrency control that require system calls. Computer applications and the programmer expect these calls or interfaces to be included at compile and link time by a given compiler and linker.

Bare machine applications require system call equivalents (direct hardware interfaces) that are independent of any OS or kernel. These interfaces are directly controlled and accessed by an AO programmer. All of the above are factors to consider in determining the number of direct hardware interfaces needed for a suite of BMC applications. Some direct hardware interfaces used in BMC applications are discussed below.

A. Static and Dynamic Memory

Static memory needs depend on the size of code, data and stack needed to run a program. When an executable is created, this information is available to the programmer. Thus, for a given executable, one can specify its requirements for memory. An AO can also be designed that can read the existing memory and restructure its code, data and stack in real memory and external mass storage or network. The code image is small as there is only one AO running at a time in the machine, and applications that are related are grouped to run together.

Dynamic memory needs are however not known until run time. In a bare machine application, an AO programmer estimates the dynamic memory. Appropriate exceptions for memory can be set to manage dynamic memory; when large dynamic memory needs arise, one can use secondary storage in place of large dynamic memory. System calls similar to malloc() and free() can be designed to support dynamic memory management. One can allow the memory

 ISBN: 978-1-61208-272-1

controller to communicate with an AO and thus provide appropriate memory interfaces to manage memory in the AO. As memory technology improves and becomes cheaper, it is also conceivable to assume full address space (4GB in a 32-bit architecture) in a machine to avoid all memory management issues and provide direct control to a given AO.

B. User Interfaces

The most common user interfaces are keyboard, mouse, touch-screen and terminal screen. These resources are managed by the OS in conventional systems. In bare machine applications, keyboard interfaces are part of an AO where the keyboard interrupt code places the data in a user buffer. Similarly, mouse data is also placed in a user buffer. An AO programmer designs the code to directly interface with a keyboard or a mouse. The terminal screen is usually controlled by a video memory or its graphics adaptor. An AO programmer can directly store output in video memory or write a bare video driver to control the screen. All device drivers supporting a bare application have to be bare and provide direct hardware interfaces to applications. They cannot, as is done, when an OS is present, be hidden from the application programmer. Other user interfaces have to be handled in a similar manner to the above interfaces.

C. Network Interfaces

Most ordinary computing devices today have one wired and one wireless network interface. The device drivers for a network interface are controlled by underlying OS. Bare machine device drivers that provide direct network interfaces to an AO are needed in BMC. Instead of current OS-dependent network drivers, an AO programmer can initialize a network driver, configure relevant internal registers, and read or write to buffers and control registers. Such a design allows direct communication to applications and avoids the need for any middleware. As the drivers are now encapsulated within an AO, the network hardware is not accessible to other applications when a given application suite is running in the machine. A bare PC USB device driver and its implementation are described in [5].

D. Process Interfaces

Many computer applications require process creation, deletion and management, which are usually controlled by an OS. In Intel x86 processors, process control and state are maintained by the CPU in a task segment. Interrupt gates are used to switch from one task to another. That can be done in a bare environment since these interfaces are accessible to an AO programmer. Control of the CPU is placed in an application program for creation of a new process (or a task). The global descriptor table (GDT) and local descriptor table (LDT) entries are used by the AO programmer to control task memory. Thus, when a machine becomes bare, the CPU and tasks are managed by an AO programmer. Task management in a bare machine is much simpler than in a conventional system, and the code size is also smaller compared to an OS-supported system. A conventional Web server system may be complex and create over 7000 tasks (in an x86 box) to provide high performance [4]. Process interfaces can eventually be generalized and made available to an AO programmer for any given CPU architecture. Today's machines hide all these interfaces under an OS or some form of similar middleware.

E. File Interfaces

In conventional systems, a file system is part of the operating system. File systems use some standard specifications such as FAT32 or NTFS. Files can be transported across multiple operating systems and applications if they use standard specifications in their design. In bare machine applications, persistent data is under the control of an AO programmer and the data itself is part of an AO. Programmers can use their own file storage specification or use a standard specification to transport files to non-bare systems. One can also do a raw file system in an AO to avoid all file management complexities and hide the files within an AO (the only AO in which they are visible). This may be the most secure way to implement a file system. File transfers can also be accomplished through a network or by message passing. A given file system interface uses a bare device driver and controls the relevant device operations.

F. Boot and Load Interfaces

Boot and load facilities are usually under control of the OS and the underlying BIOS calls. In BMC, these interfaces are controlled by the AO programmer to facilitate bare machine applications. Soft and hard boot can be used to control the machine when needed in bare machine applications. These interfaces also vary across platforms; ideally, a standard boot and load mechanism to run bare machine applications across multiple CPU architectures and machines is the best solution (what is described in Section VII, is a method that has been implemented for x86 Intel CPUs).

G. Compile, Link and Library Issues

Compilers and linkers generate different formats for executables, which pose problems in loading and running bare machine applications. There is a need for homogenization in these tools to develop common bare machine applications that can run on many pervasive devices. New programming tools can be developed to compile bare machine applications using existing libraries and batch files, or new features can be added into existing Microsoft Visual Studio and Eclipse development tools to provide bare machine compilation options. Common libraries such as string operations, memory operations, locking, shared memory, message passing, and concurrency

 ISBN: 978-1-61208-272-1

control are system dependent and part of the OS libraries. However, they can be generalized and designed to run across many CPU architectures.

VI. BOOTABLE USB

In the BMC paradigm, applications are carried on a removable storage medium such as a CD/DVD or a flash drive. This device also carries a boot program to boot and load its own application object suite. A typical way to create a bootable USB is as follows. A bootable USB is created using a special tool written in C and assembly language. This tool is a batch file that runs in a DOS window. The USB is formatted for FAT32 before its use. The bootable USB should have three files as shown in Fig. 3b. The boot file is stored in the boot sector (**#0**), the prcycle.exe file is stored at **0x3be000**, and the application file (shown in Fig. 3a as shell.exe), is stored at **0x3c4000**.

The prcycle.exe file (**22, 037** bytes in size) contains assembly code to boot a bare PC, provides the user interface/menu, and facilitates the loading of AOs (in this instance, shell.exe). It enables the switching from real to protected mode and vice versa for handling low-level interfaces. It also contains, IDT, GDT, TSS and BIOS interrupts to provide the AO programmer with direct control of the CPU. This part of the application code thus plays a key role in enabling the programmer to manage the hardware resources in a bare PC. In summary, the batch file copies files onto the USB, installs a boot program, and creates a bootable USB. This entire process does not require any software other than what resides on the USB (and is thus part of the bare PC application). There is no dependence on any specialized commercial tool or software. This enables bare PC applications to be independent of any OS-related environments and tools. It is also possible to use existing boot tools to create a bootable USB; however those tools must guarantee high security if needed in a system. The approach proposed here demonstrates building bare machine computer applications in a single environment where every aspect of software development is controlled by an AO programmer with no other dependencies. This approach facilitates enhanced security to computer applications.

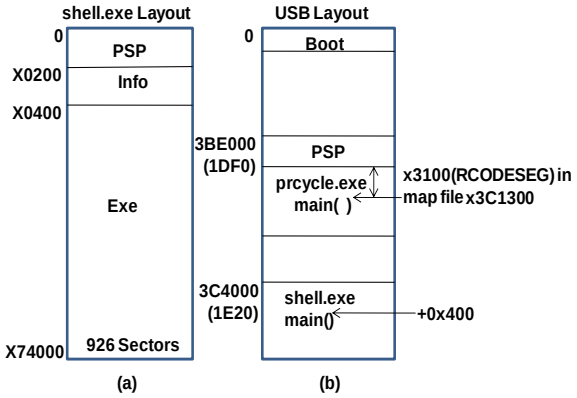

Figure 3. USB layout

VII. MEMORY MAP

As discussed in section 4, the AO programmer needs to design the real memory layout when developing a bare PC application. Fig. 4 shows a typical memory layout for a given application suite. An AO programmer prepares this map before designing a given application suite. The prcycle.exe program is used on the bare platform to load the AO at **0x600** in real mode memory. The main() entry point for prcycle.exe is located at **0x3100**, which can be obtained from the prcycle.map. When the PC is booted, it must jump to **0x3900** as instructed by this memory map. A user loads the example application (shell.exe) by using the menu provided by prcycle.exe (not shown here). The executable for this AO is loaded at **0x00111E00** as shown in Fig. 4. The reason for using this particular address for loading shell.exe is discussed below. Visual Studio 8.0 (and later editions) of compilers behave differently than the previous versions when generating an exe file. In previous versions, when the entry point in shell.map indicates **0001:00000000**, it usually implies that the main entry point in shell.exe is at **0x1000**. In newer versions, this is not the case. In Visual Studio 8.0 (C++ versions), the executable starts at address **0x400** instead of at **0x1000**. As shown in Fig. 4, the AO (shell.exe) is located at **0x00111E00**. The higher 16-bit address **0x0011** indicates that it is loaded above **1 MB** to load it in a protected mode memory address.

The lower 16-bit address **0x1E00** is derived as follows. The compiler start address for shell.exe is **0x0000**, but it actually starts at **0x400**. It was observed in the executable that the offset used by this compiler is **0x1e00** more than the actual offset in the executable. Thus, when the executable is relocated at **0x1e00**, the references to the variables were correct as it was generated by the compiler. The main entry point for shell.exe should be at **0x1e00** + **0x400** as shown in Fig. 4. A generic tool is needed to resolve such intricacies involved in generating a memory map for a mass storage device. This tool should consider compiler options, executable formats and map files to create a memory map that is suitable for a given bare machine device.

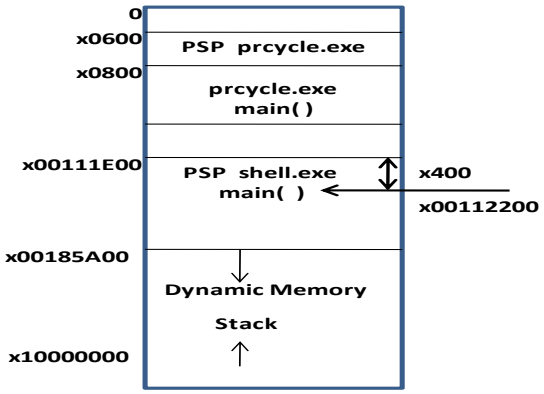

Figure 4. Memory map

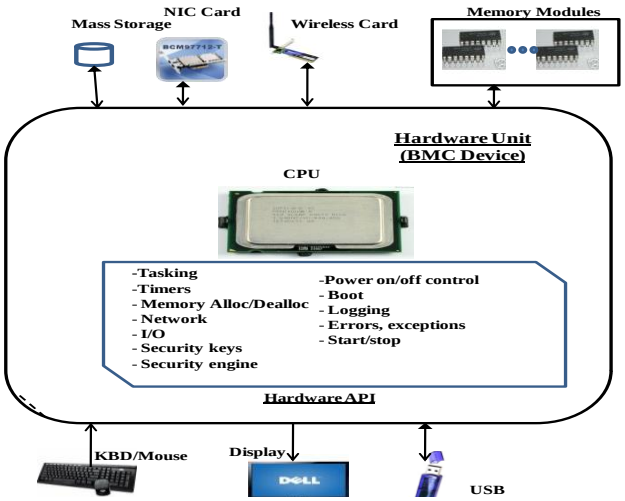

Figure 5. BMC device architecture

VIII. NOVEL FEATURES

BMC applications provide a new and innovative computer architecture that is based on current trends in technology. Fig. 5 illustrates a vision of future computing. This shows a BMC device that communicates with standard units such as memory, network card, wireless card, USB device, keyboard, mouse, display and mass storage. These units are common to many pervasive devices today. It is thus useful to write computer applications that target the BMC device as a baseline. Each device can run its own native application while using the standard hardware API as illustrated in the figure. All applications can access these interfaces and yet the hardware itself is bare. Until then, we can continue to provide these interfaces as software. The BMC architecture avoids heterogeneity in hardware, software, programming and tools.

IX. CONCLUSION

We described the BMC paradigm and showed how to build applications based on it. We identified the generic direct bare hardware interfaces needed to eliminate the OS/kernel. The BMC paradigm/approach enables these hardware interfaces to be incorporated in the hardware, thus making the latter more intelligent and able to communicate with the software. The interfaces were used to construct complex bare PC applications that have a small code footprint, are simple to use, provide high performance, and are inherently secure in design. We also presented a bare machine application architecture that enables a BMC device to be used for many pervasive applications. The new paradigm and approach will make it possible to save time, energy, and resources, while reducing the cost of developing applications for each pervasive device. The BMC paradigm demonstrates a new approach to computing based on completely self-supporting applications that eliminate all intermediary software.

REFERENCES

[1] P. Appiah-Kubi, R. K. Karne and A.L. Wijesinha, "The Design and Performance of a Bare PC Webmail Server," The 12th IEEE International Conference on High Performance Computing and Communications, AHPCC, pp. 521-526, Sept. 2010.

[2] D. Engler, "The Exokernel Operating Systems Architecture," Dept. of Elec. Eng. and Computer Science, Massachusetts Institute of Technology, Ph.D. Dissertation, 1998.

[3] FreeBSD/Linux Kernel Cross Reference, http://fxr.watson.org/fxr/source/kern/syscalls.c.

[4] L. He, R. K. Karne, and A. L. Wijesinha, "Design and Performance of a bare PC Web Server," International Journal of Computer and Applications, vol. 15, pp. 100-112, June 2008.

[5] R. K. Karne, S. Liang, A. L. Wijesinha and P. Appiah-Kubi, "Bare PC Mass Storage USB Driver," International Journal of Computer and Applications, March 2013.

[6] R. K. Karne, "Application-oriented Object Architecture: A Revolutionary Approach," 6th International Conference, HPC Asia, Dec. 2002.

[7] R. K. Karne, V. Jaganathan, T. Ahmed and N. Rosa, "DOSC: Dispersed Operating System Computing," OOPSLA, Onward Track, 20th Annual ACM Conference on Object Oriented Programming, Oct. 2005.

[8] R. K. Karne, V. Jaganathan and T. Ahmed, "How to run C++ Applications on a bare PC," 6th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD), pp. 50-55, May 2005.

[9] G. H. Khaksari, A. L., Wijesinha, R. K., Karne, L., He and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application," IEEE Consumer and Communications and Networking Conference (CCNC), pp. 803-807, Jan. 2007.

[10] G. H. Khaksari, A. L. Wijesinha, and R. K. Karne, "A Bare Machine Development Methodology," International Journal of Computer Applications, vol. 19, no.1, pp. 10-25, Mar. 2012.

[11] J. Lange. et. al, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing," 24th IEEE International Parallel and Distributed Processing Symposium , Apr. 2010.

[12] V. S. Pai, P. Druschel and Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System," ACM Transactions on Computer Systems, vol.18 (1), pp. 37-66, Feb. 2000.

[13] B. Rawal, R. K.Karne, A. L. Wijesinha, "Mini Web Server Clusters for HTTP Request Splitting," IEEE International Conference on High Performance Computing and Communications, pp. 94-100, Sep. 2011.

[14] M. Schoeberl, S. Korsholm, T. Kalibera and A. P. Ravn, "A Hardware Abstraction Layer in Java," ACM Transactions on Embedded Computing Systems, vol.10, no. 4, Article 42, Nov. 2011.

[15] S. Soumya, R. Guerin and K. Hosanagar, "Functionality-rich vs Minimalist Platforms: A Two-sided Market Analysis", ACM Computer Communication Review, vol. 41, no. 5, pp. 36-43, Sep. 2011.

[16] Windows System Call Table, Googlecode.com, retrieved Feb 16, 2012. http://miscellaneuoz.google.com/svn/winsyscall.

 ISBN: 978-1-61208-272-1