Insight Into the x86-64 Bare PC Application Boot/Load/Run Methodology

Hojin Chang
Computer and Information Sciences
Towson University
Towson, MD 21252
hchang3@students.towson.edu

Ramesh K. Karne
Computer and Information Sciences
Towson University
Towson, MD 21252
rkarne@towson.edu

Alexander L. Wijesinha
Computer and Information Sciences
Towson University
Towson, MD 21252
awijesinha@towson.edu

Abstract

Boot/load/run methodologies differ in various ways for a given operating system or kernel interfaces. They also vary for 32 bit versus 64 bit processors. Developing a boot/load/run application process for a bare PC is unique and poses daunting challenges as there is a need to understand various aspects of the CPU architecture and its internals. This paper provides intricate details of this methodology. The concepts and implementations described here can be used to construct other boot/load/run programs for pervasive devices. This paper identifies some standard API that will be useful to construct a boot/load/run process that directly runs on the hardware without a need for any middleware such as an operating system, kernel, or embedded system. It also identifies some novel ideas where this boot/load/run methodology can be made transparent to the user by moving these interfaces to on-chip. The work presented here is intended for use in development of x86-64 bare PC applications.

1 MOTIVATION

The motivation stems primarily from a need for a 64 bit boot/load/run methodology required for bare PC applications that is part of our mission in the bare machine computing research. Secondly, the current 32 bit bare PC applications developed for x86 architecture, can be ported to run on x86-64 architecture in the future. The insight into boot/load/run methodology and its implementation consequently provides a foundation for the development of 64 bit bare PC applications. The methodology presented here also has a broader impact to other pervasive devices and ultimately to the future of bare machine computing paradigm.

2 INTRODUCTION

Development of bare PC boot/load/run program methodology is not a trivial task. Most of the tools and code available on the Web assume some sort of operating system, kernel, or embedded system to use these programs. Consequently, in particular the boot process is simpler as the kernel handles complex load and run processes as needed during the execution. The traditional OS and kernel programs are mature, complex and huge that are designed to work with x86 and x86-64 architectures. They hide the architectural intricacies and complexity inside the kernel and thus often overlooked by the application programmer. For a system programmer to work with a complex kernel is a frustrating and a daunting job. In a bare PC application, the application itself manages boot, load and execution of its own program. This poses different challenges and requires handling of internals of CPU and direct communication with hardware at run time.

A bare PC application is based on an application object (AO) [4] notion and bare machine computing (BMC) [5, 7] paradigm. An AO is self-contained, self-controlled and self-executable object. An AO may consist of one or more sets of applications that are needed by user at a given time. It is a suit of end user applications that do not depend upon any execution or operating environment and run on a bare PC or a bare machine.

Numerous bare PC applications such as Web server [1], Webmail server [2], VoIP softphone [8], Split servers [3] and Transforming OS based application to bare PC [10] used x86 boot and load programs. A bare PC device driver construction for a USB is referred in [6]. A bare PC software development methodology described in [9] provides some high level concepts of developing bare PC software applications. This paper will provide insight into the boot/load/run program process that can be used for x86-64 architectures. Section 3 describes a generic methodology for building boot/load/run methodology. Section 4 describes x86-64 boot/load/run process. Section 5 shows boot/load/run implementation and API that is written in C and assembly. Section 6 shows functional operation and testing of this development process. Section 7 identifies novel features of this approach for BMC applications. Section 9 summarizes this paper.

3 METHODOLOGY

There are ample amount of resources and knowledge scattered throughout the Web to construct boot and load programs [12] for conventional platforms that are based on some sort of OS, kernel or embedded systems. As per our knowledge, there is no relevant useful links, resources, or tools available for bare PC development

978-1-880843-92-5 / copyright ISCA, SEDE 2013
September 25-27, 2013, Los Angeles, California, USA

other than our own research in bare machine computing in the past decade. This section outlines a general overview of this process and the subsequent sections deal with more specific information.

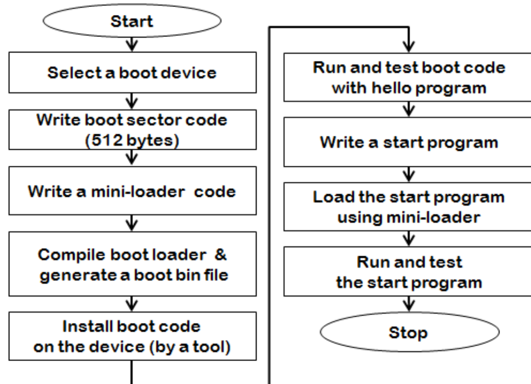

Figure 1. Methodology

Fig.1 shows a generic methodology to illustrate a boot/load/run program process. For a bare PC application, one needs to select a bootable device, which is a detachable mass storage device such as a USB flash drive. A boot sector code and mini-loader is needed that work with a bare PC. A given choice of an assembler can be chosen to write this code and compile it to generate a binary file. This bin file is transferred to a bootable USB using a boot install tool. Initial test of boot code can be done by simply printing a "**Hello**" message after the boot. Once such simple boot test is done, one can write a start program in assembly that can be loaded by the mini-loader (which is part of the boot code). The start program is the beginning of a first program after the boot process.

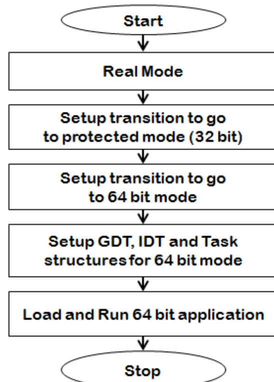

Figure 2. 64 bit Specifics

Fig.2 shows some more details needed to run a 64 bit application. When a PC is booted, it starts in a real mode, which is limited to 1MB address and has access to all BIOS (Basic Input Output System) interrupts. These interrupts can be used to accomplish simple I/O tasks that are needed during the boot and load process. The start program is used to move from real to protected (32 bit) mode where you have a full access to 4GB address space. In order to transition to 32 bit protected mode, some control registers, GDT and IDT needs to be appropriately initialized. Once the 32 bit protected mode is established, you can transition to 64 bit mode which requires different type of settings. The 64 bit mode requires setup for paging, TLB (Translation Look Aside Buffer), GDT (Global Descriptor Table) and IDT (Interrupt Descriptor Table) entries. Paging is mandatory in 64 bit mode, but optional in 32 bit mode. One can also directly go to 64 bit mode from real mode, using the same start program with appropriate setup. A 64 bit application can be loaded, run and tested for its operation once it is in 64 bit mode.

Notice that, as the 64 bit mode supports its previous real and protected modes, the boot/load/run program process for this is more complicated than its predecessor modes. The transition from one mode to another can be made using interrupt gates or long jump instruction. If we need to use any BIOS calls, we need to go back to real mode from 64 bit mode. Thus, one can design a cycle going from **real-protected-64, 64-real,** or **64-protected-real** modes. In some cases, in a 64 bit mode, if you need to run a 32 bit mode application, you may also choose to go back to 32 bit mode or run it in a compatibility mode. The compatibility mode also requires manipulation of control registers and other structures. The implementation details of this process will be described in later sections. The 64 bit mode and its usage depend upon the need of a user to run strictly 64 bit applications or a mixed set of applications.

4 BOOT/LOAD/RUN PROGRAM PROCESS

This section describes boot, load and run program process in detail. It also shows the intricacies involved for creating such programs for a bare PC application.

4.1 Boot Program

A boot program [12] usually consists of a 512 byte (could be more) block of a binary file. It is written in an assembly language and it has a single "**text**" segment. No data or stack segment is needed to create a working boot program. A boot code needs to be compiled as a binary executable. A variety of compilers such as NASM, MASM, TASM or GAS can be used to create this binary. Note that these assemblers are not compatible in their syntax and structures. A bootable mass storage device such as a hard disk, USB or a CD is used to contain the boot binary in sector 0. For an IBM compatible PC, the boot code on the bootable device is at sector 0, which is loaded into memory at **0x7c00** by its BIOS interrupt upon a power on state. The CPU starts executing this boot code

starting at this address. The boot code must contain a 2 byte signature at the end of its 512 byte block (0x55AA). A simple "**hello**" boot does not have any loader. The "hello" **text** can be printed using video memory indicating the success of a boot process.

In IBM compatible PCs, initially video memory can be used to display text on the screen. The video memory start location is at 0xB8000, which can be loaded in the ES segment register to address this memory. To display each character, it requires two bytes to be stored in the video memory, one byte for data and the other byte for the properties. Sophisticated graphics and color patterns require a graphics driver to handle the display and visualization. Simple text and graphics can be done using video memory.

4.2.1 Load Program

Detachable mass storage device (USB flash drive) is used to store boot/load, application programs and data. There is a mini-loader in the boot program that is simple and limited in its usage due to its size constraints in the boot code. When a boot code is running, the system is in real mode and it has access to BIOS interrupts. The BIOS interrupt 0x13 is used to read all needed sectors from the USB into maim memory. The load program requires the starting sector number and the total number of sectors to read from the device. Initially, it reads all sectors needed into memory during initialization and in real mode. If more data is needed to read or write to the USB in protected or 64 bit mode, a device driver is needed for USB and a sophisticated loader as well. It is also possible to fall back to real mode to read more data using interrupt 0x13 again.

4.2.2 Mass Storage and Memory Layouts

Insight into boot/load/run program process is illustrated using USB map and memory map figures as shown in Fig. 3 and Fig. 4 respectively. The USB map shows layout of files in chronological order that are installed on it using a "**USB Image Maker**" tool, which creates a bootable USB image for bare PC applications. The boot loader (BootLoader.bin) is always installed at sector 0. Sector 1 consists of an entry point code written in assembly (EntryPoint32.bin), which provides a transition from real to protected mode (16 to 32 bit). Entry point code is always loaded in sector 1.

There are three other types of binary files in the USB map as shown in Fig. 3, Mode32.bin, Mode64.bin and Hello.elf. The Mode32.bin file consists of Main.o and ModeSwitch.o objects including the 32 bit main. The Main.o (first main) object is a 32 bit C program, which demonstrates the transition to 32 bit mode. In addition, it checks available memory, obtains information using get CPUID and so on. The ModeSwitch.o written in assembly provides transition from 32 bit mode to 64 bit mode. The Mode64.bin consists of EntryPoint64.o and **Main.o** (second main) objects including the 64 bit main. The EntryPoint64.o is an assembly program, which provides entry point to 64 bit mode. The **Main.o** (second main) object is a 64 bit C program, which demonstrates the transition to 64 bit mode. The **Hello.elf** is the user application program for 64 bit mode, which is installed at the end of all other files. The total number of sectors illustrated in this example include: 106 sectors (**1** boot sector, **1** entrypoint sector, **5** Mode32.bin sectors, **65** Mode64.bin sectors and **34** Hello.elf sectors).

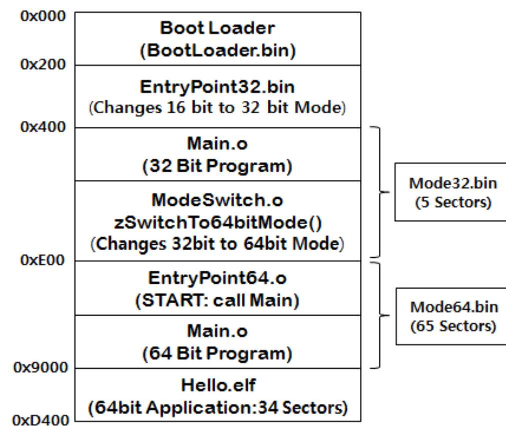

Figure 3. USB Layout

During boot code execution, in this given example 105 sectors are loaded into memory excluding the boot sector (i.e. 0xD200 bytes). The USB map shows the individual file sizes and their locations. The mini-loader loads all these sectors initially using interrupt 0x13 in real mode.

In this example, the 105 sectors are loaded into memory as shown in Fig. 4. We load these sectors starting above 1MB address as they can be accessed in protected and 64 bit modes. The starting address for EntryPoint32.bin is at 0x00010000. Thus, the boot loader jumps to the START label in this entry point code using a long JMP instruction (**jmp 0x1000:0x0000**). The ES register is loaded with 0x1000, which translate to 0x00010000, that is 1MB starting address. At this point, all segment registers have zero values. The entry point code setup transition from real to protected mode and then jumps to a C program main() using another long JMP instruction (**jmp dword 0x18:0x10200**). The GDT entry setup for this Main.o code is 0x18 (0x18/8 = 3), which is a 3rd entry in the table. The Main.o object invokes ModeSwitch.o object, which setup 64 bit mode using zSwitchTo64bitMode() function. After 64 bit mode setup it jumps to EntryPoint64.o START label using a long

JMP instruction (**jmp 0x08:0x10c00**). The GDT entry setup for Mod64.bin code is 0x08 (0x08/8 = 1), which is a 1st entry in the table. The 0th entry in the GDT table is defined as a null GDT. This entry point code will call a 64 program Main.o (second main) which provides a user menu to load and run a 64 bit application. This is part of Mode64.bin as shown in the memory map.

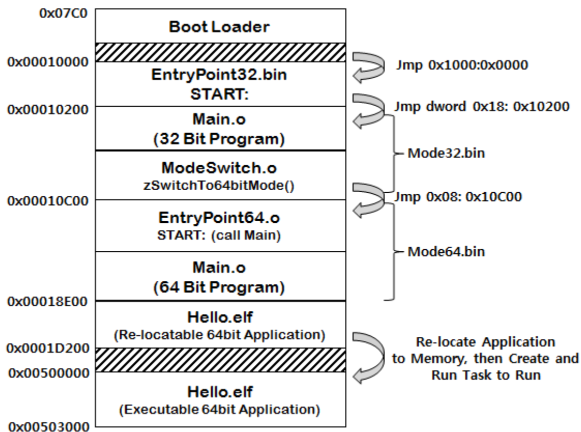

Figure 4. Memory Layout

The user menu helps to load and run the "**Hello.elf**" application. Initially, the "Hello.elf" application was loaded in memory at location 0x00018e00 during the boot time. When this application is actually loaded by user in a 64 bit mode, it may be relocated to other part of the memory such as 0x500000 as illustrated in the memory layout. In a bare PC setup, usually memory map is designed and controlled by the AO programmer and usually it is a fixed location to load a given AO. However, a given AO can have a set of applications constituted as a single user AO.

4.2.3 USB Image Maker

We built a "**USB Image Maker**" [11] tool using a C program that runs on Linux. This tool uses a "makefile" to compile its code using some input options. The tool uses all the binary and boot images as shown in Fig. 3. To install them on the USB, we use Win32DiskImager which is a free development software [13]. The USBImageMaker makes the USB bootable image, but the Win32DiskImager create a bootable USB. The USBImageMaker tool is modified so that the boot record bytes 5, 7, 9 to pass information to the bare PC programs. The byte 5 shows the total number of sectors including the boot sector (106 sectors), byte 7 indicates the size of Mod32.bin (5 sectors) and byte 9 shows the size of Module64.bin (65 sectors). The size of Hell.elf can be derived with the above parameters. This tool can also be used to pass many other parameters to the bare PC application. Notice that the bare PC boot image is not same as other OS boot images and the unused bytes in the boot image can be used as parameter space. The bare PC boot image is also not compatible with other OS boot images. Most of the code used for this tool is off-the-shelf [11] and some modifications are made to suit our bare PC development environment. More details of this tool are not covered in this paper due to space constraints.

5 BOOT/LOAD/RUN IMPLEMENTATION AND API

This section shows some internal details of implementation and some generic API that can be used to build your own 64 bit bare PC applications. Fig. 5 shows the entry point code (EntryPoint32.s), which is invoked from the boot loader as shown in Fig. 4. The snippets of assembly code are shown in Fig. 5 to illustrate the steps implemented in this program. It uses BIOS interrupt to set A20 bit, which is needed to transition to higher memory above 1MB. It initializes GDT register and loads control register 0 to go to protected mode. Finally, it uses a long jump to go to Main.o (first main) located at 0x10200 and GDT entry 0x18 (3rd entry).

```
<EntryPoint32.s>
; Activate A20 by BIOS service
  mov ax, 0x2401       ; set A20 activation service
  int 0x15             ; call BIOS interrupt service

  lgdt [ GDTR ]  ; set GDTR structure

  mov eax, 0x4000003B
 ; PG=0, CD=1, NW=0, AM=0, WP=0, NE=1, ET=1, TS=1, EM=0, MP=1, PE=1
  mov cr0, eax         ; write them to CR0, then swithch to the protected mode

[BITS 32]              ; 32 bit mode(Protected Mode)
PROTECTEDMODE:
```

Figure 5. EntryPoint32.s

The Main.c (first main) in Fig. 6 lists the main steps taken in this program written in C. It creates a page table for 64 bit mode and uses CPUID instruction to obtain the vendor-id and 64 bit mode supportability feature. Finally, it calls zSwitchTo64bitMode() assembly call that is located in EntryPoint64.s.

```
<Main.c (first main())>
//print a string on display
void zStringLinePrint( int cX, int cY, const char* cptrString );
//check memory size if it is more than 64Mbytes or not
BOOL zCheckEnoughMemory( void );
void zInitializePageTables( void ); // Create page table for 64bit(IA-32e) mode
// get CPU-ID, vendor string
zGetCPUID( 0x00, &dwEAX, &dwEBX, &dwECX, &dwEDX );
// check up supporting 64 bit or not
zGetCPUID( 0x80000001, &dwEAX, &dwEBX, &dwECX, &dwEDX );
// switch mode 32 bit to 64 bit, then jump to 64bit(IA-32e) mode
```

Figure 6. Main.c (first main())

The EntryPoint64.s program written in assembly is shown in Fig. 7. It initializes data segments with GDT

selector 2 (0x10 = 16/2 = 2). It initializes the stack and calls Main program (this is the second main).

The Main.c (second main) has many functions to run as shown in Fig. 8. This functions or API shown in this figure is not in chronological order. At the end of this program, it invokes a "**Menu**" that provides a lean user interface for loading and running 64 bit applications.

<Entrypoint64.s>

```
mov ax, 0x10  ; set 64 bit
mov ds, ax    ; DS
mov es, ax    ; ES
mov fs, ax    ; FS
mov gs, ax    ; GS
; stack for 64bit mode
;  with 1MB size
mov ss, ax      ; SS
mov rsp, 0x6FFFF8
; set RSP
mov rbp, 0x6FFFF8
; set RBP
call Main
 ; 64 bit main
 ; program
```

<Figure 7>

<Main.c(second main())>

```
void zInitializeConsole( int iX, int iY );
void zGetCursor( int *piX, int *piY );
void zSetCursor( int iX, int iY );
void zInitializeGDTTableAndTSS( void );
void zLoadGDTR( QWORD qwGDTRAddress );
void zLoadTR( WORD wTSSSegmentOffset );
void zInitializeIDTTables( void );
void zLoadIDTR( QWORD qwIDTRAddress);
zCheckTotalRAMSize();
zInitializePIC();
zMaskPICInterrupt( 0 );
zEnableInterrupt();
TCB* zCreateTask(…);
void zInitializeKeyboard(void);
BYTE zGetCh( void );
void zClearScreen( void );
void zMenu(void);
void zWaitCommand(void);
void zChoiceToStopTask(void);
int zMemDump(void);
```

<Figure 8>

<more details of selected interfaces>

void zInitializeGDTTableAndTSS(void);
- Update 64 bit GDT Code and Data (0x08, 0x10)
- The new entries reflect 64 bit needs
- Initialize TSS for 64 bit (it's different from 32 bit)

void zLoadGDTR(QWORD qwGDTRAddress);
- lgdt [rdi]
- rdi contains GDT table address

void zLoadTR(WORD wTSSSegmentOffset);
- ltr di
- di contains TSS segment(16 bytes) descriptor offset

void zInitializeIDTTables(void);
- create IDT table
- add IDT entries

(**e.g. Keyboard**: zSetIDTGate(&(ptrIDT[33]), kISRKeyboard, **0x08**, IDT_FLAGS_**IST1**, IDT_FLAGS_PGR, IDT_TYPE_INTERRUPT);)

void zLoadIDTR(QWORD qwIDTRAddress);
- 0x08 is for entry of GDT code
- lidt [rdi]
- rdi contains IDT table address
- only one IST table used(IST1)

Figure 9. Selected Interfaces Details

Most of the API shown in this figure is self-explanatory. Some important interfaces are shown in Fig. 9 with some key details. The zInitializeGDTTableAndTSS() function updates 64 bit GDT with new parameters required in 64 bit mode. It initializes TSS for 64 bit, which is different from 32 bit mode. The zLoadGDTR() function loads the GDT register with new GDT table. The zLoadTR() function loads the task register. The zInitializeIDTTable() initializes the IDT table. A sample IDT entry for keyboard is shown to illustrate this API. Notice that the keyboard IDT entry is pointing to 0x08 which is a 64 bit mode code segment descriptor. The zLoadIDTR() function initializes IDT table and loads IDT register. Only one IST (interrupt stack table) is used in our demonstration, which is used to create and run one 64 bit mode application. In 64 bit mode, task switching is done in software, whereas in 32 bit mode, it is done in hardware. The details of task management are not shown in this paper due to space limitations. All the functions described in this section illustrate a common API that can be used to construct boot/load/run programs for any x86-64 based bare PC applications.

6 FUNCTIONAL OPERATION AND TESTING

The application program, boot and loader is installed in a USB (mass storage device). This USB can be used to boot/load/run on any x86-64 compatible PC without any hard disk or OS. We used a 2GB Verbatim USB and a 64 bit Laptop (ASUS N43S Intel Core i5, 3rd generation CPU) to test this boot/load/run program process. When a power is turned on, it boots, loads its 64 bit application and runs the application. When it is booted, it starts in real mode and moves into protected and subsequently to 64 bit mode to run the application. When the system transitions to 64 bit mode, it provides a user interface (menu) to load a 64 bit application and runs. It also provides a menu option to dump memory contents for debugging purposes. We have tested a small 64 bit application program using this boot/load/run program process. This system can be expanded to run any set of applications and other features. The snap shots of the running process for the 64 bit "Hello" program are shown in Fig. 10 thru Fig. 13. The Fig. 10 shows the menu interface, Fig. 11 shows the hello program display, and Fig. 12 shows the trace produced by the USB Image Maker tool.

```
(0) Show All Tasks
(1) Load an Application
(2) Run an Application
(3) Memory Dump
(4) Test FPU Function
(5) Stop Application
```

Figure 10. Menu Interface

```
Hello, Welcome 64-bits world!!
```

Figure 11. Application Output

7 NOVEL FEATURES

This paper presents a complete methodology of creating a boot program, load application into memory and run a program without any need for operating system, kernel, or embedded system. There is no middleware required other

than the application itself. The CPU structures GDT, IDT, TSS, Video Memory, Keyboard and Display are directly controlled by the application object programmer. This programmer has a complete knowledge of application at static and run time. There is no commercial or vendor software involved in this software engineering paradigm. An end user application suite can be carried on a mass storage device and run it anywhere on a bare PC. The direct hardware control and interfaces (API) shown in this paper provide a complete overview to construct a standalone 64 bit application. Eventually, this API implementation can be moved into the processor chip thus making it intelligent and providing direct access to the programmer. We can eventually make this API standard across many pervasive devices to create portable applications that run on many pervasive devices. When hardware devices are made bare, they can be placed anywhere without a concern for its computer protection other than a physical damage or vandalism.

```
./UsbImageMaker.exe 00.BootLoader/BootLoader.bin 01.Mode32
/Mode32.bin 02.Mode64/Mode64.bin 04.Application/hello.elf

   [INFO] Copy boot loader to image file
   [INFO] File size is aligned 512 byte
   [INFO] 00.BootLoader/BootLoader.bin size = [512]
          and sector count = [1]
   [INFO] Copy protected mode program to image file
   [INFO] File size [2690] and fill [382] byte
   [INFO] 01.Mode32/Mode32.bin size = [2690]
          and sector Count = [6]
   [INFO] Copy IA-32e mode program to image file
   [INFO] File size [32948] and fill [332] byte
   [INFO] 02.Mode64/Mode64.bin size = [32948]
          and sector count = [65]
   [INFO] Copy 64bit Application to image file
   [INFO] File size [17256] and fill [152] byte
   [INFO] 04.Application/hello.elf size = [17256]
          and sector count = [34]
   [INFO] Start to write all information
   [INFO] Total sector count except boot loader [105]
   [INFO] Total sector count of
          protected mode program [6]
   [INFO] Total sector count of 64-bit application [34]
   [INFO] Image file create complete
```

Figure 12. Image Maker Trace

8 CONCLUSION

This paper presents a novel methodology to write bare PC applications that are independent of any operating system, kernel, or embedded system. It described internal details of implementation for boot, load and run process for a 64 bit application. These detailed code snippets and prototypes can be used to implement your own system to create bare PC applications. It also showed its functional operation and testing of a boot/load/run program process. Some significant contributions of this paper are identified which has a broader impact in developing future bare PC or machine applications.

9 REFERENCES

[1] 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.

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

[3] B. Rawal, R. Karne, and A. L. Wijesinha. "Mini Web Server Clusters for HTTP Request Splitting," 13th International Confrence on High Performance Computing and Comunication (HPCC), 2011.

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

[5] R. K. Karne, K.V. Jaganathan, T. Ahmed, and N. Rosa, "DOSC: Dispersed Operating System Computing," 20th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Onward Track, pp. 55-61, 2005.

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

[7] U. Okafor, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, "Eliminating the Operating System via the Bare Machine Computing Paradigm", The Fifth International Conference on Future Computational Technologies and Applications, Future Computing 2013, Valencia, Spain.

[8] 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.

[9] 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.

[10] U. Okafor, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, " A Methodology to Transform an OS-based Application to a Bare Machine Application, The 12th IEEE International Conference on Ubiquitous Computing and Communications (ICC-2013), Melbourne, Australia, 16-18 July, 2013.

[11] Snghun Han, Principles and Architecture of 64 bit Mutli Core OS, Published by HANBIT Media, Inc., Seoul, Korea, 2011.

[12] Booting, http://en.wikipedia.org/wiki/Booting

[13] Win32 Disk Imager, http://sourceforge.net/projects/win32diskimager/.