An API for Bare Machine Computing Applications

S. Almutairi
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
salmut2@students.towson.edu

R. Karne
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
rkarne@towson.edu

A. Wijesinha
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
awijesinha@towson.edu

H. Chang
Department of Mathematics & Computer Science
Susquehanna University
Selinsgrove, PA 17870
changh@susqu.edu

R. Almajed
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
ralmaj1@students.towson.edu

H. Alabsi
College of Business, Mathematics & Science
Bemidji State University
Bemidji, MN 56601
hamdan.alabsi@bemidjistate.edu

W. Thompson
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
wvthompson@towson.edu

N. Soundararajan
Department of Computer & Information Sciences
Towson University
Towson, MD 21252
nsound1@students.towson.edu

***Abstract*—Conventional computing systems require some form of an OS or kernel to run applications. These include minimal OSs and small kernels such as tiny Linux kernels, and embedded OSs. In a bare machine computing (BMC) system there is no OS or kernel, thus enabling applications to be completely self-supporting while eliminating OS vulnerabilities and overhead. These applications, which are currently written in C/C++ with some assembly code, run on desktops or laptops with Intel x86 CPUs, and integrate the necessary BMC software and hardware interfaces. This paper provides internal details of several such BMC interfaces showing how they are designed, implemented and integrated with OS-independent applications. BMC interfaces could be adapted in future for use in database servers for Big Data applications, controllers for software defined networking, and IoT devices or smart phones with a view towards improving security and performance.**

Keywords—bare machine computing, application programming interface, operating systems, Intel x86, network applications, hardware interfaces

I. Introduction

Conventional computing is based on some form of an operating system (OS) or kernel that provides an execution environment to run applications. Bare machine computing (BMC) [1] is an alternative to conventional computing that allows applications to run without requiring any OS components. BMC applications communicate directly to the hardware by means of a hardware API, and a software API consisting of commonly used functions. Our goal in this paper is to provide sufficient details of the existing BMC APIs and their usage in applications so that future implementers can extend or adapt these interfaces enabling them to be used in new domains.

A BMC application, which is referred to as an application object (AO) [2], is self-supporting in that all of the code needed for it to run is included in the AO. There is thus no OS, kernel or intermediary software in the BMC approach making it different from minimal OSs such as bare metal systems, tiny Linux kernels, and embedded OSs. BMC systems retain the advantages of such minimal systems including better performance and simplicity. In addition, BMC applications provide more security since they have no OS vulnerabilities, do not support dynamic linking, and their small size enables easier analysis for security flaws. A bare machine consists of CPU, memory, and the necessary I/O. There is no permanent storage such as a hard disk in the machine. BMC Web servers [3], mail servers [4], SIP-based VoIP systems [5], gateways with NAT, IPv6-v4 translation and IPsec [6][7], file systems [8], and SQLite databases [9] have been previously built by using the BMC APIs. At present, BMC applications run on Intel x86 desktops or laptops that serve as a bare machine. However, it is possible to transform a BMC application to run on the ARM architecture [10]. The BMC APIs could be used in IoT devices or smart phones by adapting them to work with a given hardware architecture. A bare machine can also be used as an IoT gateway, a database server to support Big Data applications, or a controller in a software defined network.

II. Conventional versus BMC Interfaces

In conventional computing, system calls from system libraries are linked with the code at link time enabling applications to run. In case of an API instead of system calls, API libraries are linked at link time as well. In either case, the interfaces are controlled by the underlying OS at execution time. Also, it is generally not possible to use such interfaces as is with a different OS or platform. Fig. 1 illustrates conventional computing and compares it with the BMC approach. In a BMC application, the API becomes a part of the application program/code and is compiled statically. There are no system calls or system libraries to be linked with the program. The hardware interface is also compiled with the application program and one monolithic executable is created that runs on a bare machine. BMC applications run

independently without a need for a centralized control or middleware, and handle events as they occur without any interruption or task switching. The BMC APIs are used during the application development phase and consist of 113 hardware and 149 software interfaces. A given application only includes the interfaces it needs.

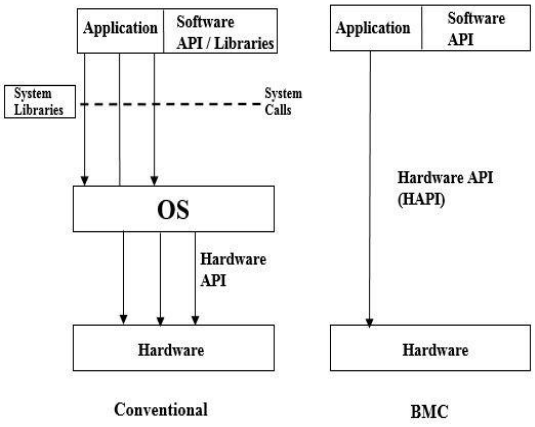

Fig. 1. Conventional computing and BMC interfaces.

III. HARDWARE INTERFACES

The BMC hardware interfaces provide an API, which allow the hardware to be directly accessed from an application program. The interfaces are described below with relevant code snippets to illustrate their usage or implementation aspects. Although the hardware interfaces given here are specific to the Intel architecture, similar hardware interfaces could enable IoT applications to run without using an OS or kernel.

A. User Input and Display

1) Keyboard: The function AOAgetCharacter() is used to read a character synchronously. This function is used to implement other functions including AOAgetHex() to read hex digits, and AOAgetString() to read a string. There is also an asynchronous interface AOAgetCharacterBuff(), where the application does not have to wait for the user to enter a key. The functions to read a character from the keyboard are used by an application as shown in Fig. 2. The internals of getting a single character synchronously are implemented using the assembly program shown in Fig. 3.

```
a = io.AOAgetCharacterBuff();  //asynchronous
a = io.AOAgetCharacter();      //synchronous
```

Fig. 2. Reading a character from the keyboard.

```
getcharasm32 PROC C public uses ebx ecx edx esi es, cursorp: dword
             mov     ax, RDataSel  ;real data segment
             mov     es, ax
             mov     eax, GKKQ     ;char buffer
             mov     esi, eax
             xor     eax, eax
             mov     eax, cursorp
             mov     dx, ax
             push    dx
             mov     bx, 2
             mul     bx
             pop     dx
             mov     ecx, eax
WaitChar:
             mov     bl, BYTE PTR es:[GKHead]
             mov     bh, BYTE PTR es:[GKTail]
             cmp     bl, bh
             je      WaitChar
             xor     ebx, ebx
             mov     bl, BYTE PTR es:[GKTail]
             add     esi, ebx
             mov     al, byte ptr es:[esi]
             cmp     bl, 254
             jz      RoundChar
             inc     bl
             jmp     UpdateChar
RoundChar:
             mov     bl, 0
UpdateChar:
             mov     BYTE PTR es:[GKTail], bl
             push    eax
             cmp     al, 13
             jz      NDisp
             mov     ebx, ecx
NDisp:
             inc     dx
             cmp     dx, 2000   ; cursor check
             jl      CursorOK
             jmp     Charout
CursorOK:
             mov     ax, dx
Charout:
             ret
getcharasm32 ENDP
```

Fig. 3. Reading a character synchronously.

2) Display: BMC display uses video memory (0xb8000 real address) and the cursor position, to support multiple text-only screens. When the current screen copies to the main memory screen area, the current screen becomes the next screen. The UP and DOWN arrows are used to move to higher or lower numbered screens. The display consists of a 25 x 80 screen layout as shown in Fig. 4 so that 2000 characters can be displayed on each screen. Each character requires 2 bytes of memory space, one byte for the character code and the other for specifying its characteristics. Graphics and images are not supported at present. At a given time, only one screen is displayed in the display. Fig, 5 shows how an application writes hex and text data onto the screen using the appropriate print functions.

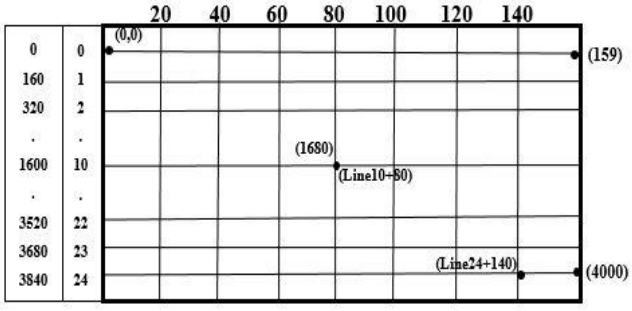

Fig. 4. Display layout.

```
retcode = io.AOAprintHex(value, Line20+20);//print 32 bit hex value
retcode = io.AOAPrintText(svalue, Line21+20);//print text
retcode = io.AOAprintCharacter('X', Line22+20);//print a single character
```

Fig. 5. Writing hex and text.

B. Device Drivers

1) Audio: An audio driver for the AC 97 chip set is written for BMC applications. This driver provides an API that includes several functions including record() and play(). A VoIP application that uses this API is given in [11].

2) NIC: NIC drivers are integrated with the BMC application program. In BMC network applications described later, the API for an Intel Gigabit Ethernet NIC is directly used by an application to receive and send packets.

3) USB: BMC applications use USB flash drives for mass storage. In addition to mass storage read and write operations, plug-and-play capability requires interfaces to detect the presence of the USB device. The BMC application directly manages the USB controller and the device. The driver provides a linear block address enabling the application to do read and write operations as in Fig. 6. The USB interfaces can also be used to perform RAW read and write operations directly. Current BMC device interfaces are designed for USB 2.0, which can be easily extended to USB 3.0. Such interfaces could be adapted to work with USB connected IoT devices.

```
retcode = usbo.WriteOp(lbaw1, writeaddress,
                BLOCK_SIZE, taskindex,
                    tasknumber[taskindex]);
retcode = usbo.ReadOp(lbaw1, readaddress,
                BLOCK_SIZE,  taskindex,
                    tasknumber[taskindex]);
```

Fig. 6. USB access to read and write.

C. Memory

1) Main Memory: BMC applications use main memory above the 1 MB area and are limited to 4 GB in 32 bit mode. Large arrays and structures are declared using dynamic memory. There are no direct calls to main memory. Instead, the application developer defines a memory map within 4 GB and designs the memory layout based on the needs of a given application. Thus, there is no centralized memory management in BMC applications. The example in Fig. 7 illustrates how an application uses dynamic memory. If memory to store 5000 integer values is needed, first an available memory slot for this data is found within the memory map. Then a start address and max value for the data is defined.

2) GDT (Global Descriptor Table): The GDT is created in real memory. All required descriptors are stored in the GDT. Each entry in the GDT provides access to memory and checks the base and limit address for each memory access. This descriptor is 8 bytes long and follows the encoding specified in the Intel Hub Controller manual. GDT entries can be read or written as needed using the provided API.

```
char *ptr;
ptr = (char*) ADDR;
// ADDR is chosen in the memory map
For (i=0 ; i < Length ; i++)
      ptr[i] = 0;
```

Fig. 7. Using dynamic memory.

3) TSS (Task State Segment) and Registers: A task running in an Intel x86-based system requires a Task State Segment (TSS) to be defined in memory. This block contains entries as shown in Fig. 8a. The example in Fig. 8b shows how to use the API to access a given entry and change the EIP (program counter) through the TSS entry. Given a task id, it is seen that we can calculate the TSS and a location for the EIP. BMC applications can directly access TSS and modify entries as needed since there is no intermediary OS or kernel. Likewise, general purpose registers, segment registers, control registers, and the IDTR, GDTR, and TR registers can be also be directly accessed by BMC applications.

	GDT Entry	Offset
1	Link	0
2	Level 0 Stack Ptr	4
3	Level 0 Stack Sel	8
4	Level 1 Stack Ptr	12
5	Level 1 Stack Sel	16
6	Level 2 Stack Ptr	20
7	Level 2 Stack Sel	24
8	CR3	28
9	EIP	32
10	EFLAGS	36
11	EAX	40
12	ECX	44
13	EDX	48
14	EBX	52
15	ESP	56
16	EBP	60
17	ESI	64
18	EDI	68
19	ES	72
20	CS	76
21	SS	80
22	DS	84
23	FS	88
24	GS	92
25	LDTR Sel	96
26	I/O MAP	100
27	0	104
28	0ff	108

```
int AOAProtected::AOAchangePC(int addr, int taskid)
 {
   int *pcloc = 0;
   int avalue = 0; //current EIP in TSS
   //EIP is at 0x20, but add 8 bytes which is
   // used to store 8 byte descriptor = 0x28
   pcloc = (int*) (TSS_ADDR + taskid * TSS_SIZE + 0x28 -
                                   ADDR_OFFSET);
   avalue = *pcloc; //current EIP value
   *pcloc = addr;
   pcloc = (int*) (TSS_ADDR + taskid * TSS_SIZE + (0x48+8) -
                                   ADDR_OFFSET);
   *pcloc = 0xc0;  //ES zero selector
   return avalue;
 }
```

Fig. 8. a. TSS block. b. API to access TSS entry.

4) Shared Memory and Stack: Shared memory, which is reserved, is located at 0x8600. Each entry in the shared memory area is 4 bytes long. There are many shared variables stored in this area that can be accessed by using the API. AOAgetSharedMemory() and AOAsetSharedMemory() are calls that can be invoked anywhere in the application program. Each task in BMC also has its own stack area. This stack area can be accessed to debug programs or change the program flow via the API.

D. Interrupts, Modes, and Timer

1) IDT (Interrupt Descriptor Table): There are 256 interrupt descriptors in the IDT. Each entry consists of 8 bytes data that can be accessed or modified using the API. All interrupts in the Intel x86 architecture have a calling function in case an interrupt occurs at a given vector location. These interrupt vectors are stored in the interrupt descriptor table (IDT), which is located in real memory in the IDT segment. There is an "intexception" class which provides code enabling the application to directly handle interrupts. The code snippet in Fig. 9 shows how an application can get control of interrupt processing and then execute new code as needed. Most hardware interrupts are processed using C++ code, with a few requiring implementation in assembly.

```
//get interrupt vector address
void (__thiscall IntException::*pFunc90)(long) = &IntException::IntException090;
   void* p90Ptr = (void*&)pFunc90;
   Function_Address_Array1[90] = (int)p90Ptr;
   //set the address in the IDT entry
   for (i = 16; i <= 114; i++) //102 entries
      {
      io.AOAsetISR(INTSELBASE, i,(unsigned long)Function_Address_Array1[i]);
      }
 //implement the ISR
void IntException::IntException090(long value)
{
   int retcode = 0;
   io.AOAPrintText("090", Line0+6);
   //ANY OTHER CODE CAN BE INSERTED HERE
   //.....
   _asm{sti}
   io.AOAstiTimer();
   io.AOAISRExit();  //special exit for ISR
};
```

Fig. 9. Interrupt processing.

2) Modes: There are two CPU modes, real and protected, in BMC systems. Mode switching is done in an assembly program by using an interrupt. Some API calls automatically switch modes to provide services. For example, INT 13H, which works in real mode, is used to access an external storage device to read a sector. The following interfaces illustrate how applications use mode changes to read and write sectors: io.AOAreadSectors(addrx2, START_SECTOR, SECTORS_TO_READ); io.AOAwriteSectors(addrx1, START_SECTOR, SECTORS_TO_WRITE);

3) Timer: A hardware timer and associated API is provided for BMC applications. For example, the current system time is obtained by calling io.AOAgetTimer(). This timer is initialized in the beginning of the program to obtain a clock ticks of 0.25 milliseconds. Clock granularity can be changed during initialization.

IV. Software Interfaces

The software interfaces consist of programs that are necessary to support bare PC applications or functions that implement commonly used operations. The necessary interfaces are included with a given application and the application directly calls the functions.

A. Loader

The loader program in Fig. 10 shows the steps for transferring the bare application executable from the USB flash drive into main memory. This code is only invoked once after boot process. The bare loader could be adapted for IoT devices according to the computing environment, which may be SoC (system on a chip) or microcontroller-based.

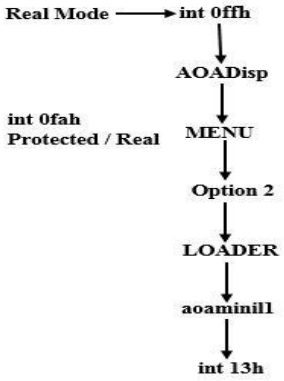

Fig. 10. Loader flow.

```
//Send Data
    x = EO.TDLPointer + EO.SendInPtr * 16 - ADDR_OFFSET;
          //check if the TDL pointer in the range
          if ((x+ADDR_OFFSET) > (EO.TDLPointer + (NO_OF_TDL-1) * 16))
              return -22;
           p1 = (long*)x;
              //now check if this data buffer is within the limits of TDL data buffers
           if ((*p1) >= (EO.TDLDataPointer + NO_OF_TDL * T_BUFFER_SIZE))
              return -23;
           send_buffer = (char*)*p1; //address from DPD pointing to next available slot
           InPtr = EO.SendInPtr;
           EO.SendInPtr++;    //this is like an InPtr which inserts packets
           if (EO.SendInPtr == NO_OF_TDL)
           EO.SendInPtr = 0; //circular list
           //add TCP header in front of the packet
           send_buffer = send_buffer + 14 + 20 - ADDR_OFFSET;//add header before data
           TCPPack_size = FormatTCPPacket(send_buffer, destIP, destPort,
                          Flags, 0, 0, 0, data, 0, 0, 0, 0, currenttask);
          // this is to format IP packet
          // ip and EO object instantiations can be avoided by placing the code in this object
          send_buffer = send_buffer - 20; //20 byte IP header
           retcode = ip.FormatIPPacket(send_buffer, TCPPack_size, destIP,
                          TargetMAC, TCP_Protocol, delegflag, delegateip, currenttask);
          send_buffer = send_buffer - 14; //14  byte ethernet header
          SendCountTotal++;
          retcode = EO.FormatEthPacket(send_buffer, TCPPack_size+20, IP_TYPE ,
                        TargetMAC, InPtr, sendtype, delegflag, delegatemac, currenttask);
//Receive Data
RXSize =   EO.ReadData(&Data, &PacketType, ip.msourceIP, macaddr);
retcode = ip.IPHandler((char*) Data, RXSize, macaddr, starttime, Current_Task);
if(protocol == TCP)
    retcode = tcp.TCPHandler(&IPPack[HeaderLength],(TotalLength-HeaderLength),
                        &IPPack[12], &IPPack[16], protocol, macaddr, starttime, currenttask);
```

Fig. 11. TCP send and receive.

B. Network Applications

Network applications in conventional systems are written in a variety of programming or scripting languages. Each language defines its own syntax/API (the socket interface) which is used by applications to access the TCP/IP protocols implemented by the underlying OS or kernel. In contrast, BMC network applications are written in C/C++, and use the BMC APIs to directly access the NIC, and the necessary network

protocols. For example, the code snippets in Fig. 11 illustrate send and receive for a TCP application using an Ethernet NIC. The application sends and receives data directly from the Ethernet buffers and manages the circular lists for send and receive. The relevant TCP, IP and Ethernet interfaces are FormatTCPPacket(), FormatIPPacket(), and EO.FormatEthPacket() for send, and EO.ReadData(), IPHandler() and TCPHandler() for receive. Security if needed can be provided via IPsec interfaces [7], TLS interfaces [12], or SRTP interfaces [13] (for VoIP applications). Similar APIs could be used in IoT applications, where network connectivity will depend on device capability and may require OTA (over-the-air) updates or a gateway.

```
void DBProcess()
{
 while(runState)
  {
   //process a given request
   Retcode = processRequest();
   suspend(delay); //return to main task
   }//end of while loop
   completeDBTask(); //task is done delete
}//end of process
 //get task address
 Function_Ptr_X = &apptask::DBProcess;
 Function_Address_Array[4] = (long)getFunctionPtrOthers(Function_Ptr_X);
 // store funtion ptr in an array
 //100 tasks in a task pool
    for(i=0; i <100; i++)
      {
       Task_ID=createTask((long)Function_Address_Array[4],0); //create a DB task
        if(stk.full() == 0)
          {
           stk.push(Task_ID);
          }
          else
          io.AOAPrintText("test.cpp stack is full", Line24);
       }
//insert task into task list
insertDBTask();     //insert task into task list (POP())
```

Fig. 12. Database processing.

C. Tasks

In a BMC system, the application itself completely controls tasks since there is no OS or kernel. Each task is a unique function implemented in the application. The function address, which is used to create the task, is captured using a special call and stored in an array. After creation, a single task or a pool of tasks is stored in the stack. Tasks are popped from the stack, inserted into a circular task list, selected to run on a first come first serve (FCFS) basis, and returned to the stack when done. A task runs as a single thread of execution without interruption and suspends itself only when it needs to wait for a given event. The application includes a task function that stays in never ending loop. Task functions are implemented using software interfaces such as AOAcreateTask(), AOAinsertTask(), and AOAsuspendTask().

For example, to run one hundred database processes in parallel on a bare machine using tasks, define a DBProcess() function and a stack that to store these processes. The function implements the activity in a database process as a never ending loop and suspends the process when waiting for an event thereby returning control to the main task. Usually, each process will follow its own state transition diagram that is implemented using a switch case statement in the code. In the code snippet in Fig. 12, which illustrates the preceding logic, DBProcess() calls the processRequest() function to process a given request. After a given case is processed, processRequest() returns, and the task gets suspended. When the task is complete, it is deleted and the task is returned to its stack. An adaptation of the above can be used in a bare database server to handle Big Data applications in the IoT.

```
retcode = fobj.InitObj(taskindex, currenttask);
findex = fobj.createFile(flName1, &startAddress, &fsize100, att100);
int WriteFile( const unsigned int findex, const char *buffer,
               const unsigned int offset, const unsigned int count,
               const unsigned short taskindex, int ctask);
int ReadFile( const unsigned int findex, char *buffer,
               const unsigned int offset, const unsigned int count,
               const unsigned short taskindex, int ctask);
```

Fig. 13. File access to read and write.

D. File System and Database

1) File System: The BMC file system uses a USB driver and follows the FAT32 file specification. USB flash drives are used for mass storage and files. The file system is implemented as a special class called FileObj [8]. This class must be initialized before files can be accessed. Only one file task runs to serve file interfaces. A file must be created before its use, and has a file index (findex). Use of the file system API is illustrated in Fig. 13, where a BMC application uses the file index to write and read to a given file. A similar file API could be used with IoT edge systems.

2) SQLite: The SQLite database engine was transformed to run on a bare PC [14], integrated with the bare PC file system [8], and extended to provide a comprehensive mass storage system [15] for bare PC applications. After SQLite has been initialized and enabled for client requests, a user interface can be provided to access the database via the bare Web server (with HTTP, PHP and standard SQLite commands). IoT devices and smartphones used to interact with them could use bare SQLite variants to build database applications that run without the support of any OS or kernel.

E. Miscellaneous

1) Conversions: The BMC software interfaces provide many functions that applications use to convert between data types such as strings, characters, and numbers (hex, int, long etc.), and to display their values. The necessary functions are directly included in the application program (separate general-purpose libraries are not needed as in conventional computing).

2) Debugging: The function AOAExit() is used by BMC applications to return to the main menu. A memory dump can then be used to debug the program. Customizable trace functions are provided to transfer necessary data into memory for debugging purposes. A special memory area is reserved to store trace data.

V. Related Work

Many approaches to minimize the OS impact on applications exist [16]-[20]. Linux is widely used as the OS in IoT devices, with Raspbian, Ubuntu/Ubuntu Core, Debian and Android being popular choices. Many other OSs including Windows, FreeRTOS [21], RIOT [22], TinyOS [23], and Contiki [24] are also used in IoT devices. FreeRTOS is designed as a real-time OS for embedded devices, but can also run on microcontrollers. It is a minimalist system allowing static allocation. RIOT is a microkernel-based OS that targets low power wireless devices. It supports C/C++ applications and a variety of network protocols. TinyOS is an embedded OS designed for wireless sensors. The TinyOS code is statically linked with a program consisting of software components. Contiki is designed for constrained devices with network connectivity, and enables concurrent programming via protothreads. Although these minimal OSs have many features in common with BMC systems, they are different from self-supporting BMC applications, which do not use any OS or kernel.

VI. Conclusion

We described OS-independent hardware and software interfaces used in BMC applications. We also gave examples of code snippets to illustrate the use of the BMC API. While the BMC interfaces are currently used only in devices with an Intel architecture, they can be adapted to work with other hardware architectures in the future. This will enable IoT devices to run applications with no OS support. Benefits of the BMC approach include eliminating OS overhead and OS vulnerabilities. By standardizing the BMC hardware API among CPU architectures, it will be possible to run applications on any device without the need for intermediary software.

References

[1] U. Okafor, R. Karne, A. Wijesinha, and P. Appiah-Kubi, Eliminating the operating system via the bare machine computing paradigm, 5th International Conference on Future Computational Technologies and Applications (Future Computing), 2013, pp. 1-6.

[2] R. K. Karne, *et al.*, Application-oriented object architecture: A revolutionary approach, 6th International Conference, HPC Asia, 2002.

[3] L. He, R. K. Karne, and A. L. Wijesinha, The design and performance of a bare PC Web server, International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[4] G, H, Ford, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, The design and implementation of a bare PC email server, 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC), pp. 480-485.

[5] A. L. Alexander, A. L. Wijesinha, and R. K. Karne, Implementing a VoIP SIP server and user agent on a bare PC, 5th International Conference on Future Computational Technologies and Applications (Future Computing), 2010, pp. 8-13.

[6] A. K. Tsetse, A. L. Wijesinha, R. Karne, A. Loukili, and P. Appiah-Kubi, An experimental evaluation of IP4-IPV6 IVI translation, ACM SIGAPP Applied Computing Review, Vol. 13 Issue 1, March 2013, pp. 19-27.

[7] N. Kazemi, A. L. Wijesinha, and R. Karne, Design and implementation of IPsec on a bare PC, 2nd International Conference on Computer Science and its Applications (CSA), 2009.

[8] S.Liang, R. K. Karne, and A.L.Wijesinha, A lean USB file system for bare machine applications, 21st International Conference on Software Engineering and Data Engineering (SEDE), 2012, pp. 191-196.

[9] W. Thompson, R. K. Karne and A.L. Wijesinha, Interoperable SQLite for a bare PC, 13th International Conference Beyond Database Architectures and Structures (BDAS), 2017, pp. 177-188.

[10] A. Peter, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, Transforming a bare PC application to run on an ARM device, IEEE Southeastern Conference (SECON), 2013, pp. 1-6.

[11] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, A peer-to-peer bare PC VoIP application, 4th IEEE Consumer Communications and Networking Conference (CCNC), 2007, pp. 803-807.

[12] A. Emdadi, R. K. Karne, and A. L. Wijesinha, Implementing the TLS protocol on a bare PC, 2nd International Conference on Computer Research and Development (ICCRD), 2010.

[13] A. L. Alexander, A. L. Wijesinha, and R. Karne, An evaluation of Secure Real-time Transport Protocol (SRTP) performance for VoIP, 3rd International Conference on Network and System Security (NSS), pp. 95-101.

[14] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal, Transforming SQLITE to run on a bare PC, 7th International Conference on Software Paradigm Trends, 2012, pp. 311-314.

[15] W. V. Thompson, *et al*., A mass storage system for bare PC applications using USBs, International Journal on Advances in Internet Technology, vol. 9, no. 3 and 4, 2016. pp. 63-74.

[16] D. R. Engler and M.F. Kaashoek, Exterminate all operating system abstractions. Fifth Workshop on Hot Topics in Operating Systems, USENIX, 1995, p. 78.

[17] 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, 2010.

[18] GitHub – ReturnInfinity/BareMetal-OS, https://github.com/ReturnInfinity/BareMetal-OS, [Accessed 1-15-19].

[19] Linux Kernel Tinification, https://tiny.wiki.kernel.org/, [Accessed 1-15-19].

[20] A Minimal Rust Kernel, https://os.phil-opp.com/minimal-rust-kernel/, [Accessed 1-15-19].

[21] The FreeRTOS Kernel, https://www.freertos.org/, [Accessed 1-15-19].

[22] RIOT, https://www.riot-os.org/, [Accessed 1-15-19].

[23] TinyOS Home Page, http://webs.cs.berkeley.edu/tos/, [Accessed 1-15-19].

[24] Contiki, http://www.contiki-os.org/, [Accessed 1-15-19]