Design Issues in Running a Web Server on Bare PC Multi-core Architecture

Nirmala Soundararajan, Ramesh K Karne, Alexander L Wijesinha and Navid Ordouie
Computer and Information Sciences
Towson University
Towson, USA
e-mail: {nsoundararajan, rkarne, awijesinha, nordouie}@towson.edu

Hojin Chang
Math and Information Science
Susquehanna University
Selinsgrove, USA
e-mail: changh@susqu.edu

***Abstract*—We consider the design and implementation of a bare PC Web server with no OS or kernel running on a multicore architecture. Previous work has demonstrated initialization, loading and running of a 32-bit web server on a single core in a multicore configured system. The main design issues that need to be addressed are balancing the load, designing re-entrant code, enforcing concurrency control, partitioning network logic, sharing the network interface and designing multi-tasking execution. We describe a novel bare PC Web server architecture and design for addressing these issues. We also provide initial performance measurements that demonstrate the feasibility of this architecture and its implementation. It is shown that with this design and implementation, the main bottleneck impeding multicore parallelism is using a single Ethernet card in the system to handle multiple cores. This work serves as a basis for identifying issues that may exist in other networking and multicore configurations for a bare PC Web server.**

***Keywords*— Bare Machine Computing, Bare PC, Intel x86, Multi-core, Web server.**

I. Introduction

Bare Machine Computing (BMC) is a two-prong approach. First, a computer is bare without any hard disk or resident operating system or kernel. Secondly, it is a different programming paradigm, where a given application suite and its programs directly communicate with the hardware. There is no middleware running in the machine. The applications and system programs are homogenized to run as a single program. The application suite is designed to perform only the intended functionality and nothing else. When a communication is made between two BMC machines, it is the most secure way to operate as it avoids all OS vulnerabilities. The computing box has no ownership or added value. Anyone can use the box anywhere. The BMC paradigm is applicable to any pervasive computing device. This is an alternate way of computing for some chosen closed applications that are distinct from all other OS based universe. It is evident that BMC applications can be used for defense and other closed secure communications but may not be immediately used for commercial world where there is more focus on open systems and the global world.

The elimination of OS abstractions was proposed [1] over twenty years ago. Since then, similar ideas have been in research focus including virtualization [2], tiny OS [3], Palacios and Kitten [4], IO-Lite [5], OS-Kit [6], bare metal Linux [7] and more. Two other systems are often mixed up with bare machine computing. LibOS [19] is used to link kernel OS libraries in user space to provide more flexibility to applications. Bare metal servers [20] offered by IBM uses “bare metal” term in a different context, indicating that it is running a single tenant at a time. The term is also used to distinguish it from modern forms of virtualization and cloud hosting. These systems still have OS or kernel dependency during the operation. These references are shown to illustrate the concepts; however it is obvious that they were explored long time ago. These approaches fall in the middle of a spectrum where a full-blown OS is on one end and the BMC at the other end. The BMC being the extreme end of this spectrum, one can’t go beyond this to achieve full control of an application with the ultimate security and simplicity. There are other architectures such as Raw [8], which focus on parallelizing customized applications, that are based on using compilers at run time to map to the hardware. These approaches are different from the BMC paradigm and its objectives.

There are numerous BMC applications cited in the literature. Some examples of these applications include: Web mail server [10], Email servers [9], VoIP [11], SQLite based mail [12] and more. These applications run as multi-threaded programs with hundreds of threads and yield high performance with high security as there is no centralized OS or kernel. These applications inherit very short image with limited functionality and a single programming environment with no outside dependencies and controls. It is essentially a closed end user system with full control by the user. This paper will further explore the work done before in web servers running on bare PC with multicore capabilities [13]. Increase in web server performance was observed in multicores due to flow-level parallelism in web server workloads [14] in Windows OS. Tunings to web server software, network interrupt processing and OS (Linux) scheduling were attempted in [15] to allow scaling of web servers in multicore architectures. Some key factors that affected the web server performance were attributed to memory footprint, control of shared resources and setting core affinity to the loads [16] while using 2 representative workloads, one serviced by cache and another that required significant I/O. Address bus utilization increased

[19] https://lwn.net/Articles/637658/

[20] https://en.wikipedia.org/wiki/Bare-metal_server

978-1-7281-7303-0/20/$31.00 ©2020 IEEE
DOI 10.1109/COMPSAC48688.2020.0-195

and saturated on the cores [17] in the experiments carried out with each Network Interface Card (NIC) dedicated to each core on a multicore web server and was contended as the main bottleneck in the web server performance. Attempt to increase the scalability of Apache web server [18] to efficiently utilize multicores was made by using multiple listening sockets to bind to a single port on a host.

This paper is based on the design of a webserver on a 4-core multicore architecture with a single NIC card and is organized as follows. Section II outlines the integrated protocol for web server including: HTTP and TCP. Section III describes design issues related to multicore implementation for the web server. Section IV describes the novel architecture used to implement web server that addresses the above design issues. Section V present novel design and implementation features. Section VI provides some performance measurements of the web server on multicore architecture. Finally, Section VII narrate the conclusion.

II. Integrated Protocol

A web server serves client requests using HTTP protocol. Internally, TCP/IP/Ethernet provide the communication between a client and a server. As the BMC paradigm has no layering, the HTTP and TCP protocols are integrated as shown in Fig. 1.

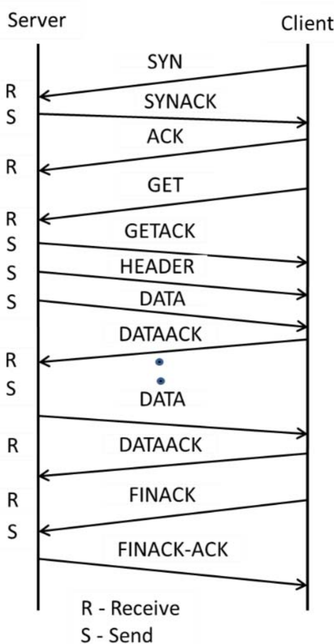

Fig 1. Server and Client Interfaces

The BMC programmer has complete knowledge and control over the Ethernet controller and its driver. In the Ethernet driver, there are two circular lists one for transmit and another one for receive as shown in Fig. 2. This implementation has up to (4096) entries in the lists. Some network cards have limitations on the size of these lists. These lists are situated in user memory. However, in BMC, all memory is in user memory.

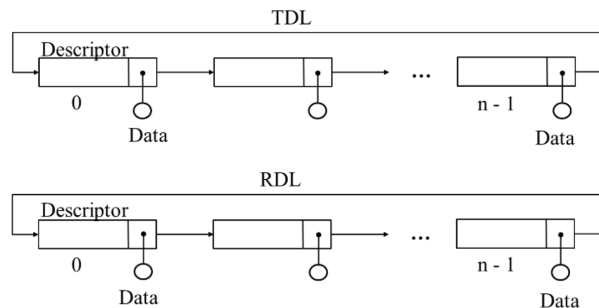

Fig.2 TDL and RDL Structures

Each entry has a descriptor and a pointer to the data. The descriptor consists of status and other control information. The software can check the status in the descriptor and the hardware can write the status. There can be a race condition, if both hardware and software come there at the same time. The Transmit Descriptor List is referred to as TDL and the Receive Descriptor List is referred to as RDL. Notice that the Ethernet driver and the card can operate in full duplex mode, where receive and send can happen concurrently.

The above description of the integrated protocol and Ethernet buffers is relevant to this paper and the web server implementation. In BMC web server, there are three basic task structures, Main Task (MT), RCV Task (RT) and HTTP Tasks (HT). There is one MT, one RT and many HT's. When a program starts execution, it begins with the MT. When a packet arrives, MT gives control to RT to run and process the packet arrived. When it is time to run a HTTP request, the HT runs to send data and update the TCP state. For each client request, there is a separate HT. The RT task suspends when a received packet processing is complete. The HT suspends after sending data to a client. When FINACK comes from a client, the HT gets terminated and returns to a task pool. Fig.3 illustrates this task state transitions.

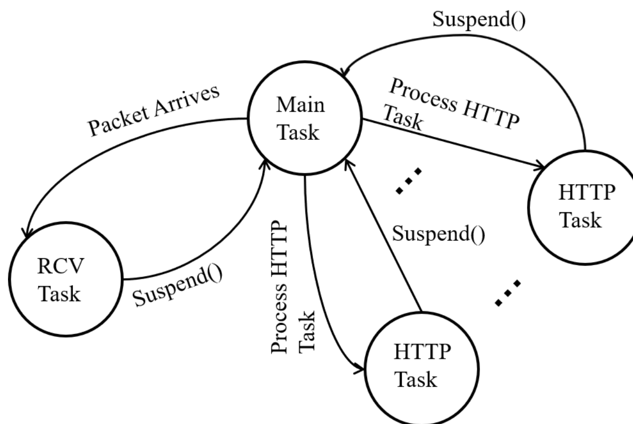

Fig.3 Task States in BMC

The IP and Ethernet protocols simply validate and remove their headers before passing the packet to TCP object. When a client sends a request to a server, the packet is captured by NIC and stored in RDL. Similarly, when a packet needs to be transmitted, it is placed in the TDL. Fig.4 shows the actual

steps involved when a packet arrives. All these steps run as a single thread of execution in the machine.

```
//check if D bit is set, Download Complete
1. EO.isRDescDone(EO.ReceiveOutPtr))  //Main Task
     //Read received data                         //RCV Task
2.RXSize =   EO.ReadData(&Data, &PacketType, ip.msourceIP, macaddr);
  //call IP Object
3.retcode = ip.IPHandler((char*) Data, RXSize, macaddr, starttime,
    Current_Task);
    //call TCP Object
4.retcode = tcp.TCPHandler(&IPPack[HeaderLength],(TotalLength-
  HeaderLength),&IPPack[12], &IPPack[16], protocol, macaddr, starttime,
  currenttask);
    //process TCP Packet
     //first time, call this function
5.retcode = ListenHandler(TCPPack, size, SourceIP, TargetIP, macaddr, timer,
    tcp->Flags, Protocol, currenttask);
  //after SYN, call this function
6.retcode = OtherHandler(TCPPack, size, TCBRecordNum, timer,
   currenttask);
    //update TCB Table
```

Fig 4. Single Thread of Execution

The MT checks if a packet is arrived by probing the "D" bit in the Ethernet descriptor. If the "D" bit set, then it calls (step2) to read the data. When the data is read from the RDL entry, it calls step 3 to remove IP header and handles to TCP object (step4). The TCP object processes the packets, if it is a SYN packet, then it handles it to a ListenHandler() function, otherwise, it handles the packet to an OtherHandler() method.

The RT tasks continuously runs through this single thread and after updating the status of the request in the TCP table, it returns to the MT. Notice, during this processing, CPU is busy and not wasting any time and not switching to other tasks. These two methods provide all the necessary functionality needed in the TCP object.

III. Design Issues

In BMC, there are many design issues to be considered for optimal design and implementation. Some of these issues are discussed here.

A. State of HTTP Requests

At a given point in time, there are many concurrent requests coming from clients. Each request goes through state transitions as shown in Fig. 1. We must maintain the state of each request and respond to the clients accordingly. This state is maintained in a table known as TCB (TCP Control Block). The responses to a client will be based on the state of a given request. Hence, we give a unique TCB number (TCBNO) to each request and then track that request as it goes through its various states.

B. Addressing TCB Table

In order to address TCB table, we must make each entry unique. The most common way to make the entry unique is using a Hash Table. The Hash Table index is usually derived from a client's IP and Port number. Hash tables also require covering for collisions by using a linked list. To avoid hash table, we used an indexing technique to optimize the design. Each client's port numbers are limited to 64K, we define a Port Table (PT) in the system. When port numbers collide with other clients, we then use a linked list structure to store the collided entry. The linked list is implemented as a memory structure to avoid address pointers. The port table points to the linked memory structure to keep the collided entries.

C. Network Interface Card (NIC)

In our desktop there is only one NIC on the motherboard. There are receive and send circular lists that interface between NIC hardware and Ethernet software. In our multicore environment, there are four cores Boot Strap Processor and Application Processors. These four cores are designated as BSP, AP1, AP2 and AP3. The BSP core assumes execution when computer starts. We used BSP as the core for communicating between NIC and Ethernet driver. The send and receive parts of the data are managed by BSP.

D. NIC Send Buffering

There are two circular buffers in the Ethernet device driver that manage send and receive data as shown in Fig. 2. As there are four cores that can send data asynchronously, we need four send data buffers to hold the data until it is placed in the send circular list. The BSP manages the send data buffers and periodically checks and places the data in the send circular list. Notice, BSP can also process HTTP requests, thus there is a need for four send data buffers.

E. BSP Buffering

As the BSP manages all the receive data for HTTP requests, this data is destined for the appropriate core which is processing a HTTP request. As the cores run concurrently, we need communication buffers to send arrived packets from Ethernet to appropriate cores. We need communication buffers between BSP and appropriate HTTP request processing cores. Notice, BSP can also process HTTP requests, thus there is a need for four communication buffers. We preferred this method as a means of communication between the cores instead of interrupts to avoid interrupt overhead.

F. Synchronization and Locking

As the cores access buffers asynchronously, there is a need to lock a buffer while the other core is accessing. In our case, we need synchronizing locks for send and communication buffers. These locks must be appropriately set and reset.

G. Process Lists

As each core can execute HTTP requests in multi-processing manner, we need a linear list that keeps all the task requests corresponding to their client requests. To avoid search in this list, we used indexing by keeping a TCBNO table that keeps the index for the linear list.

H. Load Balancing

As AP1, AP2 and AP3 have the same processing work, we simply used a round robin approach to balance the load on cores. We used the same round robin for dispatching requests when BSP also participates in processing HTTP requests. Load balancing based on CPU utilization can be a good

strategy to use, but to keep the design simple we adopted the round robin approach for now.

I. TCP Processing

The major processing of HTTP request involves processing the request at TCP level and maintaining the TCB table. We divide this TCP processing into two main components. ListenHandler() and OtherHandler() functions. The ListenHandler() receives a SYN packet and sends a SYNACK. The OtherHandler() takes care of processing the rest of the HTTP/TCP protocol. Thus, the load balancing algorithm is placed at BSP when SYN packet arrives. Although each core has its own TCP object, only BSP uses ListenHandler() code and BSP and APs can use OtherHandler() code. Each core also has its own TCB table. The TCB state in the BSP is SYNRCVD state and other cores have the current state of a given request.

J. Deleting State

The FINACK response from the client deletes the state in process list, TCB table entry and PT and the whole request is relinquished from the system.

IV. ARCHITECTURE

As shown in Section 3, multicore architecture poses many design challenges. Fig. 5 illustrates a novel architecture for implementing multicore bare machine web server. Throughout the architecture, search is avoided and replaced with index tables. Although index tables take more space, it is affordable in BMC implementation. A client's port number is used as an index into a PT. Each entry in the PT will refer to a record with a structure that has TCB record number, port no and other miscellaneous attributes. Multiple clients may use the same port number thus causing collisions in the PT. To address this problem, a linear list of port number records is stored in an array as shown in the architecture. The main purpose of the PT is to provide a TCB no that is unique for each client's request.

The TCB no is used for many other purposes in the architecture. It is used to access a TCB record that contains the state of the client request (basically the transitions shown in Fig. 1). The structure stored in the TCB Table contains about 400 bytes of information. The TCB no is also used to access a task that is running on behalf of a client request. The client request tasks are stored in a linear array as shown in the Fig.5. The linear array index is stored in the Task Index Table (TIT) which is addressed by a TCB no.

The BSP, AP1, AP2, AP3 in a four-core architecture can process any client request. However, BSP acts as a load balancer between network interface card (NIC) and cores. The BSP performs NIC interface and is responsible to receive and send the packets. In addition, it can also process HTTP requests. Accordingly, BSP runs MT, RT tasks and HT task in addition if scheduled, whereas other cores only act as HTTP request processors and run only HT Task. Due to the BSP architectural role as a centralized processor, it receives and transmits all packets to the NIC and uses buffering for send and receive operations to control concurrency issues. In addition, BSP is also chosen to load balance requests when a SYN packet arrives. A simple round robin strategy is used to balance the clients load. When other types of packets arrive for a given request (other than SYN), BSP identifies the destination as it stored the information in PT and TCB Table. The arrived packet in BSP will be stored in an appropriate communication receive lists CLIST, CLIST1, CLIST2, CLIST3. When cores send data to NIC, they put their data in send lists DLIST, DLIST1, DLIST2 and DLIST3. Periodically, BSP polls the DLISTs to send data to the NIC. As all cores are running concurrently, appropriate locking mechanisms are required as shown in the Fig.5.

The novel architecture illustrated here was a result of many iterations and trials to obtain a simple system for BMC implementation. It is possible to implement such architecture in BMC, as there is full control of the machine from boot process to execution process. It is much harder to implement this model in an OS environment with full control.

V. DESIGN AND IMPLEMENTATION

Design and implementation of a Web server on multicore architecture using BMC paradigm poses many challenges. Multicore configuration, initialization and multicore operation was described in [13]. This section will describe some additional novelties and features that are unique to BMC architecture.

A. TCP Code Partitioning

As the BSP and AP roles are different at a network level, the TCP code is partitioned to suit their needs in BMC. A separate copy of TCP code is given for each core to tailor their code for their specific needs. This also solves the problem of re-entrant code in TCP for multiple cores. As mentioned before the ListenHandler() and the OtherHandler() functions perform most of the work done in TCP object. Load balancing, communicating with NIC is done by BSP. Each core has their own main loop to serve HTTP tasks.

B. AP Cache Enable

In order to enable cache for APs, the following code is needed as shown in Fig.6. If their cache is not enabled, they run much slower as they must fetch instructions and data from main memory.

C. BSP Control Flow

A control flow to process client's request in BSP is shown in Fig. 7. This flow shows a high-level description of its implementation in BSP main loop. As shown in Fig. 4, a packet arrival can be detected and RCV task can be invoked. When the RCV task runs, it also inserts packets into appropriate CLIST based on a partitioning algorithm. After running the RCV task, the Data List (DLIST) for each core is checked if data needs to be sent through NIC. Data packet(s) are sent if the DLIST contains any send requests. If the BSP is partitioned to run HTTP requests, then it will also process

the HTTP request as shown in Fig. 7. Otherwise, the BSP control returns to main loop processing new requests.

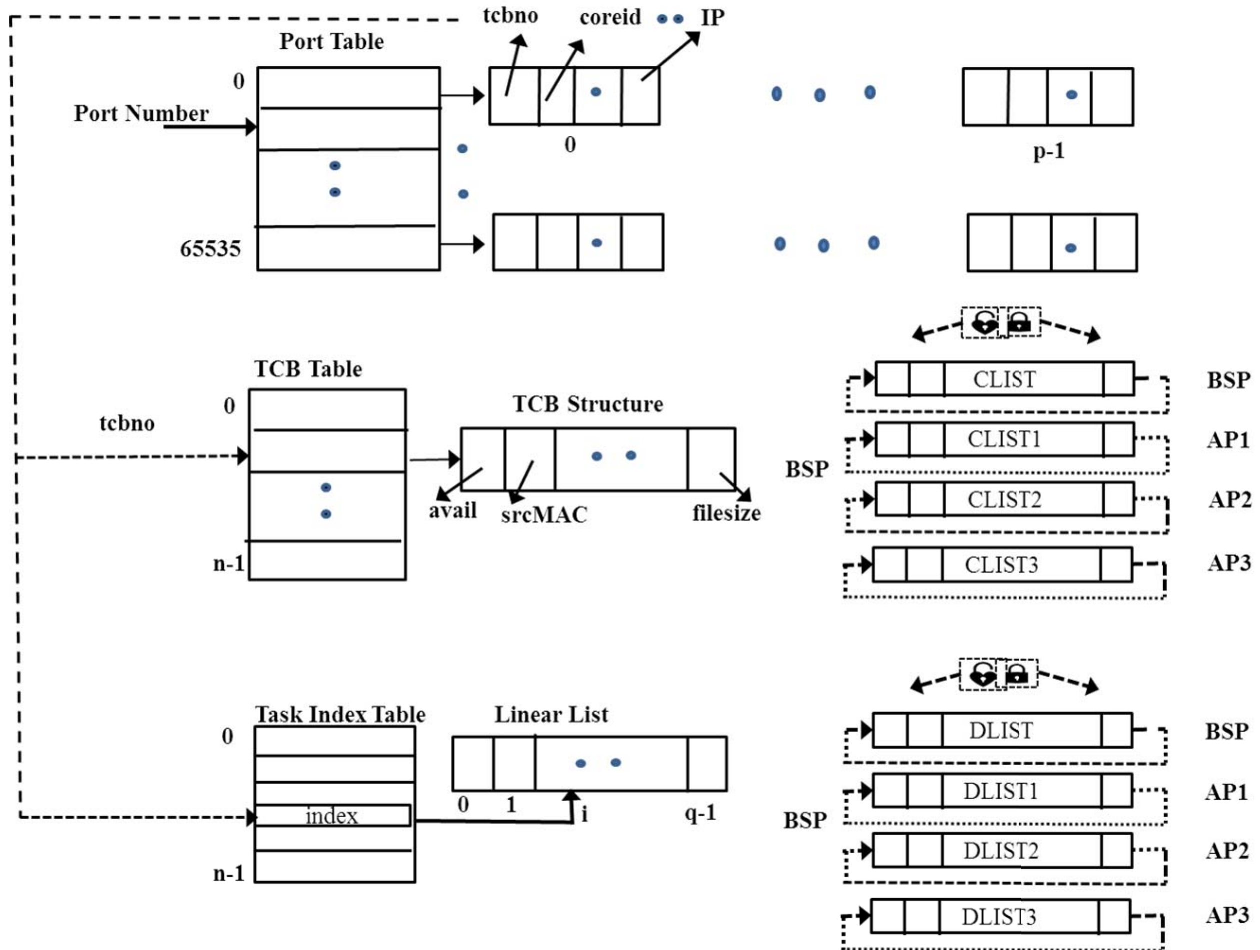

Fig. 5 Multicore web server architecture

```
zcacheEnableAP PROC C public
mov eax, cr0
and eax, 0bfffffffh ; bit 30 is reset
mov cr0, eax
ret
zcacheEnableAP ENDP
```

Fig. 6 AP Cache Enable

D. *HTTP Control Flow*

A control flow to process HTTP requests is shown in Fig. 8. The same control flow is used by BSP and APs. In APs this is the main program running in a loop. When a BSP receives a packet from a client, it checks its TCB table and finds the core-id of a given request. The core-id was inserted into the table based on a request's load balancing strategy. This core-id helps to insert the packet in an appropriate Communication List (CLIST). Each core processing a request checks its CLIST and processes an arrived packet. If it is not a valid packet type received in the CLIST, it causes an error. The processing of an arrived packet is done by calling the OtherHandler() method in the appropriate cores TCP object.

This method processes the packet and updates its status in TCB table entry. As shown in Fig. 1, when a GET request packet comes from a client, a HT is inserted into the linear list by the OtherHandler() method. Thus, when GET arrives, its task is removed from the linear list and run. Also, when time expires or FINACK arrives, its task is removed from the linear list. In addition, if a RESET comes from the client, then also its task is removed from the linear list. In all these cases, the core will run the task and return to its main loop. Notice, in BMC, the entire flow control of HTTP request processing is controlled and coded by a programmer at programming time. In conventional systems, the operating system controls this kind of flow at run time.

E. *Locking Mechanism*

As multiple cores access CLIST and DLIST entities concurrently, these sharing resources must be locked. In BMC, one can use memory addresses (that are in real mode memory) to set up lock area. These are locking and unlocking direct hardware interfaces to do these operations. An actual such code is described in Fig. 9. An instantiated interface object "**io**" represents the direct hardware interface class. This

C++ interface calls a C interface and that in turn calls an assembly call as shown in Fig. 9. This is a typical flow that is used to implement all direct hardware interfaces. Depending upon its use, one can put program logic at C++, C, or assembly level. Notice that there is a full control of real shared memory at an application level. The assembly call shows how the lock area is updated with disabling and enabling interrupts which is also under programmer control.

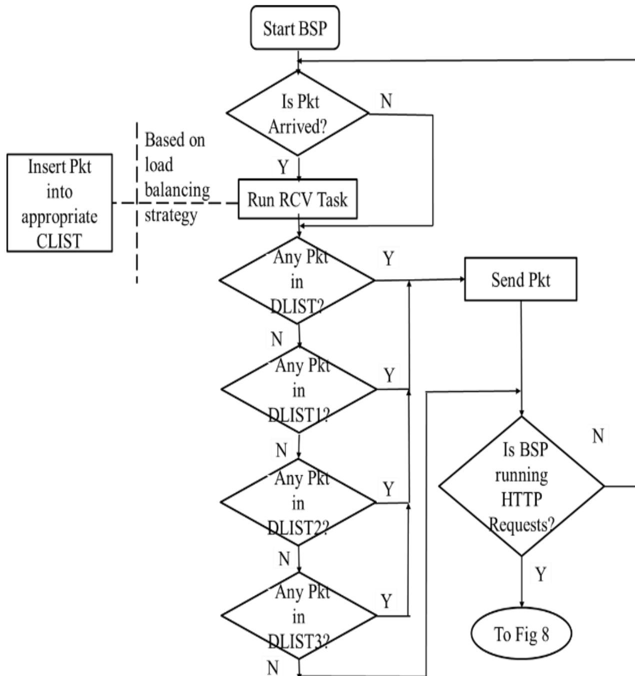

Fig. 7 BSP Control Flow

F. *Task Mechanism*

The task mechanisms implemented in BMC are novel and unique. It is completely controlled by an application programmer as shown in Fig.10a and Fig.10b. A task is defined as a function and has its own TSS (Task State Segment). The GDT (Global Descriptor Table), IDT (Interrupt Descriptor Table) and control registers can be directly controlled by its application developer. After initialization, BSP and APs run in their main loop. The main loop in each core runs HTTP tasks as needed. Fig.10b illustrates mechanisms to run tasks in cores with actual code snippets with an example of AOArunTask3() for AP3. Task implementation is in Fig.10a with an example.

Each task has a unique Task_ID (1608 in this example). This task id is used to address its TSS and a GDT entry is created to address this TSS. AOArunTask3() C++ function in turn calls a C function named ap3_runTask(). This C function in turn calls an assembly function as shown in the figure. In this example, it calls an interrupt with 0x44 (or 68) vector number. When the interrupt vector table is initialized, the descriptor related to this vector will be initialized with task gate descriptor using method set_gatedesc().

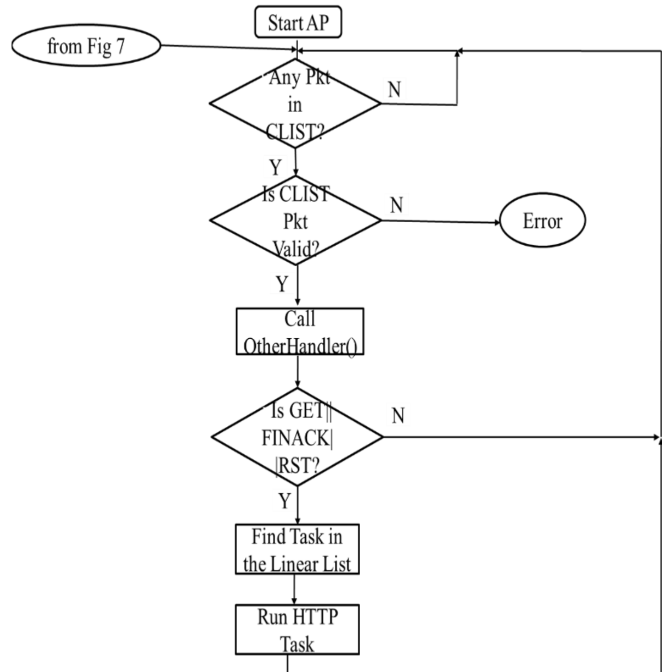

Fig. 8 AP Control Flow

Each core has the capability to run multiple HT's to process client requests.

```
retcode = io.AOAtestsetLockAPCLIST1(); //Lock DLIST1  C++
retcode = CtestsetAPLockDLIST1(offset, value);  //C
retcode = testsetlockdlist1(value);          //Assembly
testsetlockdlist1 PROC C public uses ebx edx, value:DWORD
      cli
      push   es
      push   ebx
      push   edx
      mov    ax, MEMDataSel
      mov    es, ax
      mov    ebx, S_Base
      add    ebx, 14ch
      mov    edx, value
      mov    eax, 0
      lock   cmpxchg DWORD PTR es:[ebx], edx
      pop    edx
      pop    ebx
      pop    es
      sti
      ret
testsetlockdlist1 ENDP
```

Fig 9. Locking Code

Fig.10b illustrates how task switching occur in this design. In this example, notice that 0x44 (68) is the interrupt used for AP3 tasks. Similarly, 0x43, 0x42, 0x28 interrupt vectors are used to run HT's in AP2, AP1 and BSP cores. When "INT 44h" is executed by the CPU, it goes to the IDT and obtains the GDT offset. In this case, it was initialized with 0x110 indicating that it is a 34th entry in the GDT.

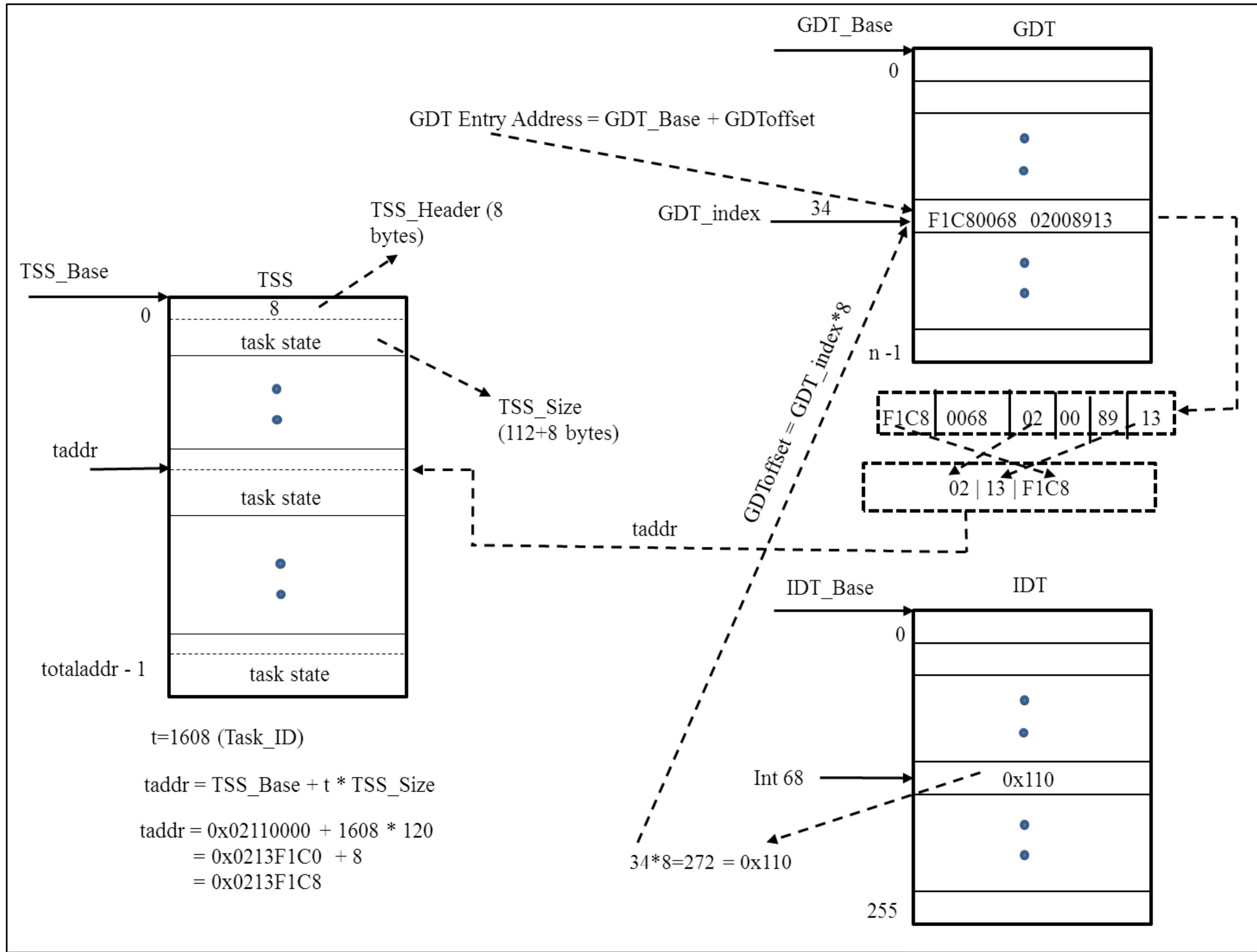

Fig.10a. Task Implementation

```
retcode = AOArunTask3(Task_ID);
// each core , Task_ID = t , C++
// using Task_ID as the index, calculate TSS address
// set GDT entry for this TSS
retcode = ap3_runTask(Task_ID);   //C
retcode = ap3_runTaskasm32(Task_ID);   //Assembly
ap3_runTaskasm32 PROC C public Task_ID:DWORD
        push eax
        mov  eax, Task_ID
        int   44h
        pop eax
        ret
ap3_runTaskasm32 ENDP
------------------------------------------------------------------------
// initialize idt
void Cmsetidtentry(int idtNo, int offset, int selector, int ar)
set_gatedesc(idt + 0x44, 0, 0x110, AR_TASKGATE);
 //34th 34*8 = 272 = 0x110
```

Fig.10b. Task Invocation

Notice each descriptor in x86 architecture takes 8 bytes. The CPU takes this offset and it addresses the GDT entry. It calculates the GDT entry address as shown in the Fig.10.a. The GDT entry consists of 8-byte descriptor in a descriptor format. This descriptor is mapped to a 4-byte address as shown in this figure. This four-byte address is taddr in the TSS table. A TSS entry consists of a TSS header and TSS data. The TSS header (8 bytes) is used for debugging purposes. The task mechanisms shown in this paper are unique to BMC implementation and it is used in many applications as part of BMC arena.

G. BMC Novel Design Features

In BMC, the design and implementation of a web server on multicore architecture differs in many ways compared to applications designed to run on operating systems. Some significant attributes of BMC that are novel and unique are listed as follows:

1) The entire memory is real memory, there is no virtual memory and paging
2) Real mode, protected mode and compatibility modes of x86 processor are under programmer control
3) Real mode memory can be used by programmer transparently

4) Interrupts are controlled by a programmer
5) Control flow is designed apriori by a programmer
6) A given program execution is controlled explicitly by a programmer
7) Tasking structure and scheduling is controlled by a programmer
8) A program is statically linked
9) Locking is part of an application code
10) All direct hardware interfaces are part of an application code
11) Application performs only intended functions
12) Concurrency control is easier to implement in BMC
13) Indexing can be used more liberally in the design due to less usage of overall memory
14) TSS, Program Counter, IDT, GDT can be redefined and controlled by a programmer

Hopefully, BMC is more secure than any other OS or kernel-based applications.

VI. PERFORMANCE MEASUREMENTS

Dell Optiplex 9010 at 3.4 GHz processor with 4 cores and 4GB memory is used to conduct measurements. 3 Bare PC Web clients, two with 3700 requests/second and one with 1000 requests/second and an additional windows client is used to collect data shown in the following graphs.
These clients make requests to the multicore web server running on multiple cores as described before. We do not have hard disk and virtual memory in Bare Machine. The size of the file request was 4KB and the network was 1 Gigabits/sec bandwidth. The data collected and the plots shown in this paper are described as follows.

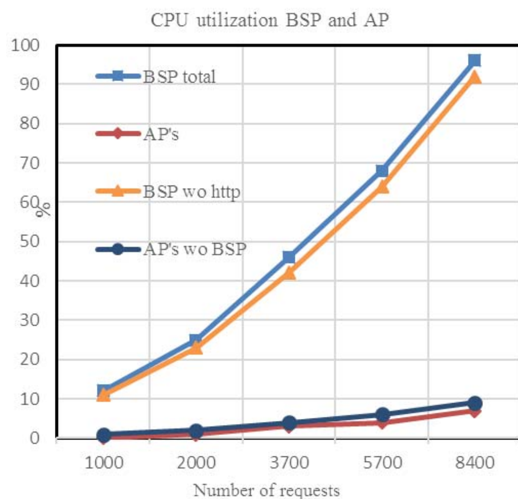

Fig. 11 CPU Utilization

Fig. 11 shows CPU utilization for all cores. BSP only handles the send and receive interface of packets with the NIC.
The charts indicate two cases, where BSP handles the HTTP load and does not handle any HTTP load. There is a minor difference in CPU utilization in these two cases. A maximum of 8400 requests per second were run on four cores where incoming requests were equally distributed on all cores.
Notice, the APs utilization is quite small compared with the BSP. This is because BSP was assigned as the main interface to the Network (Receive Task and Send Task) while APs were only doing HTTP Tasks.

Fig. 12 shows a pie chart of BSP CPU utilization with respect to receive and send processing, when there is no HTTP load assigned to BSP. The CPU utilization reached to 92% without any HTTP load. This indicates that the BSP processor is overwhelmed with the network interface handling load of sending and receiving packets.

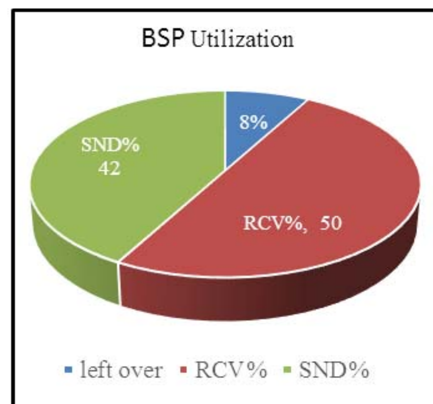

Fig. 12 BSP CPU Utilization

Fig.13 shows average and maximum time of a request that stays in a linear list before its completion.

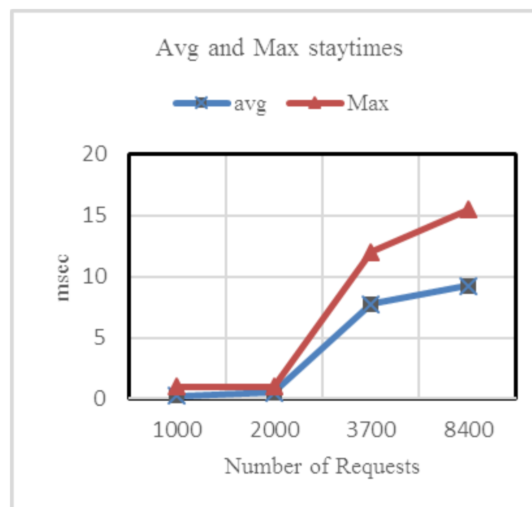

Fig.13. Stay time in Linear List

Notice that the stay time measured between a GET request and its FINACK from the client increases rapidly as the load increases.
Fig.14 shows the number of maximum requests running in parallel in all cores. When 8400 requests/sec were running,

there were about 100 requests running concurrently. This is sometimes referred to as MAX Parallel.

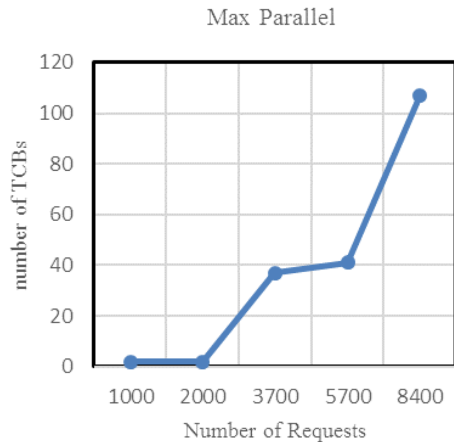

Fig 14. Max Parallel

Fig. 15 shows the first response time of server measured by Wireshark from a Windows client. As expected, the first response time increases with respect to the number of requests.

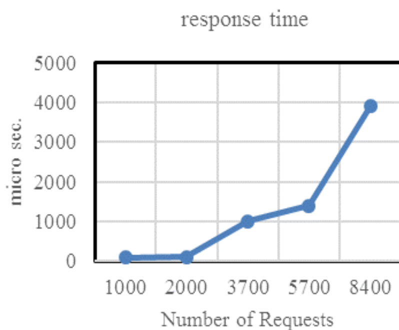

Fig. 15 First Response time

Fig. 16 shows the connection time of server measured by Wireshark from a Windows client. As expected, the connection time increases with respect to the number of requests increase.

The performance measurements indicate that the current BMC architecture has some bottlenecks beyond 8400 requests/sec. Fig.1 indicates that a client request consists of many individual steps starting from SYN to FINACK with a variable time in between interactions. It is obvious, that there is some inherent serial part in each request. We can't parallelize these requests completely due to inherent serial portion as defined by Amdahl's law. This paper focused on exploiting parallelism at inter-request levels from multiple clients. However, this parallelism was limited due to the BSP bottlenecks of handling NIC interfaces as shown in Fig. 2. Further research is needed to find new avenues to parallelize a Web server running on multicores in the BMC approach. Many commercial Web servers utilize clustering and multiple NICs to address this problem.

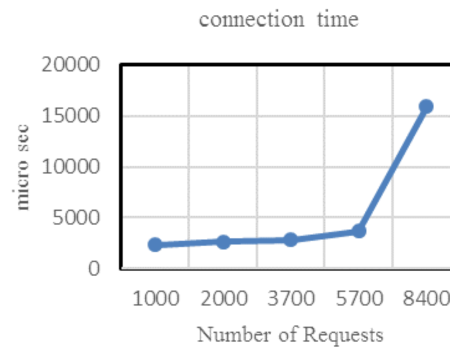

Fig. 16 Connection time

VII. CONCLUSIONS

This paper proposes a novel architecture for implementing a Web server on multiple cores without a need for OS or kernel that run on bare PCs. This paper demonstrates that the bare machine computing paradigm is applicable to multicore architectures. Novel and unique design issues related to a Web server application on bare PCs are identified and implemented. It is shown that in a bare machine computing environment, design choices such as indexing is much easier to afford and implement. Complex systems such as Web servers can be implemented on bare machines with total control by application programmer. Some novel and simple ways of implementations are described to demonstrate the potential of BMC that are only possible in bare machine computing paradigm and not applicable to OS based applications. The paper also hints that BMC based applications maybe inherently designed for security although beyond the scope of this research paper. The performance measurements illustrate that many requests were executed in parallel, however, the BSP saturation limited the scalability of this architecture. Further research is needed to address the scalability and speedup problems in the proposed architecture.

REFERENCES

[1] D. R. Engler, "The Exokernel Operating System Architecuture," Ph.D. thesis, MIT, October 1998.

[2] G. Ammons, J. Appayoo, M. Butrico, D. Silva, D. Grove, K. Kawachiva, O. Krieger, B. Rosenburg, E. Hensbergen and R. W. Wisniewski, "Libra: A Library Operating System for a JVM in a Virtualized Execution Environment," VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments, June 2007.

[3] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler, "TinyOS: An Operating System for Sensor Networks," In: Weber W., Rabaey J.M., Aarts E. (eds) Ambient Intelligence. Springer, Berlin, Heidelberg, pp. 115-148.

[4] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen and R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable

Virtualized and Native Supercomputing," Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Atlanta, GA, 2010, pp.1-12.

[5] V. S. Pai, P. Druschel and W. Zwaenepoel, "IO-Lite: A Unified I/O Buffering and Caching System," Proceedings of ACM Transactions on Computer Systems, Volume 18, Issue 1, February 2000, pp. 37 – 66.

[6] "The OS Kit Project – CS @ Utah," University of Utah, http://www.cs.utah.edu/flux/oskit.

[7] T. Venton, M. Miller, R. Kalla and A. Blanchard, "A Linux-based tool for hardware bring up, Linux development, and manufacturing." IBM Systems Journal, Vol. 44, Issue 2, 2005, pp. 319-329.

[8] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, "Baring it all to Software : Raw Machines," IEEE Computer, September 1997, pp. 86-93.

[9] G. H. Ford, R. K. Karne, A. L. Wijesinha and P. Appiah-Kubi, "The Performance of a Bare Machine Email Server," 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009), IEEE / ACM Publications, Sao Paulo, Brazil, 2009, pp. 143-150.

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

[11] G. Khaksari, A. L. Wijesinha, R. K. Karne, He, L. and S. Girumala, "A Peer-to-Peer Bare PC VoIP Application,", IEEE Consumer Communications and Networking Conference, Seamless Consumer Connectivity, Los Vegas, Nevada, 2007.

[12] H. Alabsi, R. K. Karne, A. L. Wijesinha, R. Almajed, B. Rawal, and F. Almansour, "A Novel SQLite-Based Bare PC Email Server," 15th International Conference, BDAS2019, Ustron, Poland, 2019, pp. 341-353.

[13] H. Chang, R. K. Karne and A.L. Wijesinha, "Migrating a Bare PC Webserver to Multi-core Architecture," IEEE 40th Annual Computer Software and Applications Conference, Atlanta, Georgia, 2016, pp. 216-221.

[14] F. Urem and Ž. Mikulic, ''The impact of multi-core processor on web sever performance'', 32nd International Conference for Information and Communication Technology (MIPRO), May 2009

[15] R. Hashemian, D. Krishnamurthy, M. Arlitt, N. Carlsson,'' Improving the scalability of a multicore web server", Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE), April 2013, pp. 161-172.

[16] A. S. Harji, P. A. Buhr, T. Brecht, "Comparing High Performance Multi-core Web server Architectures", Fifth Annual International Systems and Storage Conference (SYSTOR), Article no. 1, 2012

[17] B. Veal and A. Foong, "Performance Scalability of a Multicore Web Server", Proceedings of the 3rd Symposium on Architecture for networking and Communications systems (ANCS), 2007, pp. 57-66.

[18] Y. Lu, R. Shiveley, J. Ruby, "Scaling the Performance of Apache Web Server on Intel Based Platforms", White Paper Apache Web server Open Source Software Solutions, Intel.