Migrating a Bare PC Web Server to a Multi-core Architecture

Hojin Chang, Ramesh Karne, and Alexander Wijesinha
Department of Computer and Information Sciences
Towson University
Towson, MD, US
{hchang,rkarne,awijesinha}@towson.edu

***Abstract*—Bare PC applications run without the support of any OS, kernel, or embedded software. Many bare PC applications such as Web servers, mail servers, SIP servers and VoIP clients have been developed previously for the x86 Intel architecture. We migrate a bare PC Web server application enabling it to run on a 64-bit multicore desktop processor. Migration poses many challenges as this application runs on a bare PC. The 64-bit CPU architecture is significantly different from its 32-bit counterpart, making migration non-trivial. First, an attempt was made to run the existing code as is on the 64-bit processor without using any 64-bit CPU architectural features. Second, paging required by the 64-bit architecture was implemented. This requires many design changes to deal with paging and virtual memory issues since the Web server application is based on real memory without any paging. Finally, multi-tasked execution of the Web server on multicore is undertaken. This phase of migration is complex due to the complexity in 64 bit architecture and its intricacies. We give technical details underlying the migration process and present results of testing functional operations with some preliminary data to validate this approach.**

Keywords- Bare Machine Computing, Bare PC, Intel x86, Multi-core, 64-bit architecture, Web server.

I. INTRODUCTION

Bare Machine Computing (BMC) [1] is based on running computer applications on a bare machine without any resident operating system (OS). The BMC programming paradigm differs from a conventional programming paradigm in that the programmer manages all the necessary system resources. This paradigm enables full control of the system via the application software, which directly communicates with and controls the underlying hardware. BMC applications are written in mostly in C/C++ with some use of assembly language (MASM, NASM or TASM) as needed. As described in [2], one or more end-user applications can be written as a single entity referred to as an application object (AO) that runs on a bare machine. Existing BMC applications include Web servers [3], mail servers [4], SIP Servers [5] and VoIP systems [6]. BMC applications can run as multi-threaded programs with thousands of threads and yield high performance with high security as there is no centralized OS or kernel, no external dependencies and no resident mass storage.

Many low-overhead operating systems and kernels have been developed for a vari\ety of environments. For example, TinyOS is an OS for low-power wireless devices [7]. On the other hand, Kitten is a lightweight high-performance OS that can be used together with Palacios, a virtual machine monitor for applications running in a virtualized environment [8]. The elimination of OS abstractions was originally proposed in [9]. In contrast, the BMC paradigm eliminates all intermediary software providing the combined benefits of improved performance and security.

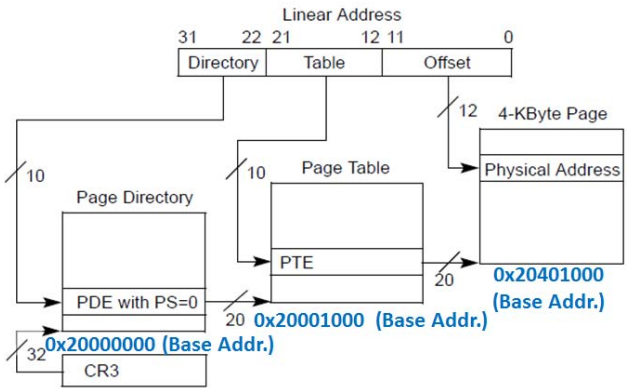

Figure 1. 4K Paging

As newer machines are based on a 64-bit processor and multi-core, 32-bit bare PC applications need to be migrated to the new environment. Many techniques have been proposed for migrating applications to multi-core in an OS environment. In [10], tools for migrating applications to multi-core based on Windows and Linux are surveyed. The use of components and partitions for multi-core migration of legacy real-time systems is described in [11]. In [12], the difficulty of migrating real-time software used in the automotive industry to multi-core is discussed and an approach for developing an automated tool based on static program analysis is suggested.

Migrating bare PC code to run on the 64-bit architecture is not the same as migrating OS-based applications. The primary difference is the absence of any OS or other intermediary software between bare PC applications and the hardware. This means that a bare PC application, regardless of whether it is 32-bit or 64-bit, has to itself manage system resources. We describe the technical details underlying the migration process for a 32-bit bare PC Web server. In particular, we consider design and implementation differences in 32-bit versus 64-bit bare PC applications with a view towards building a tool for automating the migration process in the future.

0730-3157/16 $31.00 © 2016 IEEE
DOI 10.1109/COMPSAC.2016.15

The rest of the paper is organized as follows. Section 2 outlines some basic problems in migrating a 32-bit bare PC application to run on 64-bit. Section 3 describes how paging is handled, including the related architectural and design issues. Section 4 illustrates the migration path to multicore and related implementation issues. Section 5 gives preliminary results to validate multicore functionality and its operation. Section 6 discusses issues unique to migration that are specific to bare machine computing and possible future research. Section 7 presents the conclusion..

II. BASIC ISSUES IN MIGRATION

The original bare PC Web server application [3] was written in Visual Studio 10 using C/C++ (32 bit) code and some MASM (Microsoft assembler) code. It also used TASM (Turbo assembler) and NASM (Netwide assembler) code for startup and boot code. The server ran on a single CPU x86 desktop (Dell Optiplex 520) with no paging, and no hard disk. A USB flash drive is used to store the entire application (boot, startup and application code including data). In order to migrate the 32-bit code to run on an x86_64-bit multicore desktop (Dell Optiplex 9010), the first attempt was to run the original server code without changes. Several issues were identified, some of which are discussed below.

Static variables had incorrect values in them stored in BSS (block started by symbol) although the BSS segment was not used in the 32-bit code). This did not appear to cause any problems on the 32-bit machine. Bare PC applications use batch files to compile and link i.e. Visual Studio 10 "/bin" files are used for compilation without any use of header files (or libraries). The problem was fixed by adding a "/MERGE:.bss=.data" option in the link for running on 64-bit machine. This option merges all data into a single data section. It was also found that the 64-bit machine hangs if dummy block braces "{….}" were used (without any conditions preceding). This was not an issue in 32-bit; the braces were simply removed to fix this problem. Lastly, we note that the same C++ class was compiled in two directories and linked, which should not be the case. Simply renaming the class in one directory solves this problem..

III. PAGING AND SINGLE CORE

The 32-bit bare PC code has no paging i.e., the CR0 (control register 0) PG (paging) bit and CR4 (control register 4) PAE (physical address extension) bits are zeros [14] (p.1961). As the Web server is a 32-bit application, paging mode was enabled by setting the CR0 PG bit to 1. This needed to be done before the Web server tasks are created. We therefore created a paging data structure and initialized the structure with the necessary memory before paging was turned on. The entire Web server executable is 325KB and a 1GB of address space is used for the application including code, data and stack. We used a 4K paging scheme and created a data structure as shown in Figure 1. It also shows the actual values used in the PDE (page directory entry) and PTE (page table entry) entries. The CR3 is initialized with 0x20000000 (512MB), which is the base address of page directory (PD). This initialization is done in the TSS (task

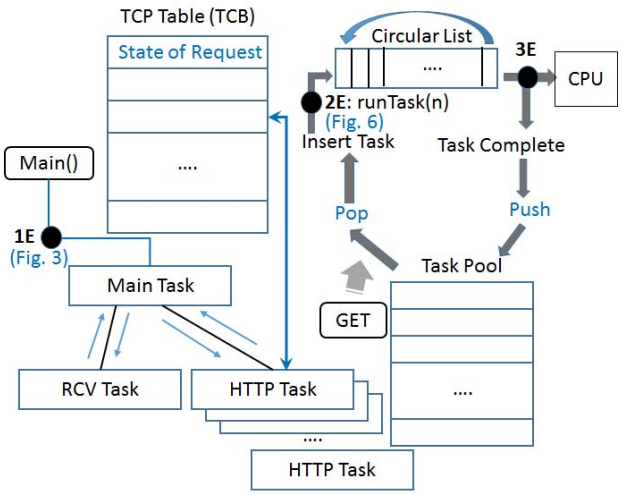

Figure 2. Current Webserver Architecture

state segment) for the Web server application. There are 1024 entries in PD (there is only one PD in the system). Each PD entry (PDE) points to a separate page table (PT), and there are 1024 PTs. Each page table entry (PTE) points to a page, where the offset is used to address the exact location in memory. The paging is managed by the CPU, and software only initializes the data structures. All entries were provided in order to address up to 4GB memory.

The 32-bit server uses physical memory up to 1GB. This memory is contiguously mapped in our paging structure, thus avoiding page replacement overhead and page faults, i.e. each logical page is mapped to one physical page. The 32-bit server also uses LDT (Local Descriptor Table) in addition to GDT (Global Descriptor Table). This causes a problem in paging mode as it needs separate paging structures for LDT. We addressed this problem by eliminating LDT and creating new GDTs for paging. As the Web server uses a single address space and there is a single AO running in the bare PC, this paging model is efficient. However, adding paging to the 32-bit code posed some implementation and testing difficulties especially when the entries have wrong values or are not initialized.

IV. MULTICORE MIGRATION

Multicore migration poses many challenges in that a bare PC application has to deal with architectural, design and implementation issues in addition to functional operation, testing, and the unique bare PC programming and computing paradigm. A bare PC Web server is a complex system that differs significantly from its OS-based counterparts. Figure 2 shows a high level view of the bare PC Web server architecture. There is a main task (MTSK), receive task (RTSK) and several http tasks (HTSK). The MTSK is always running in the system. When a packet arrives, the RTSK runs as a single thread of execution. The RTSK processes packets as they arrive and updates a table that keeps track of TCP parameters and state (known as the TCB). Each entry in the TCB maintains the status of its corresponding client request from start to finish. A task pool

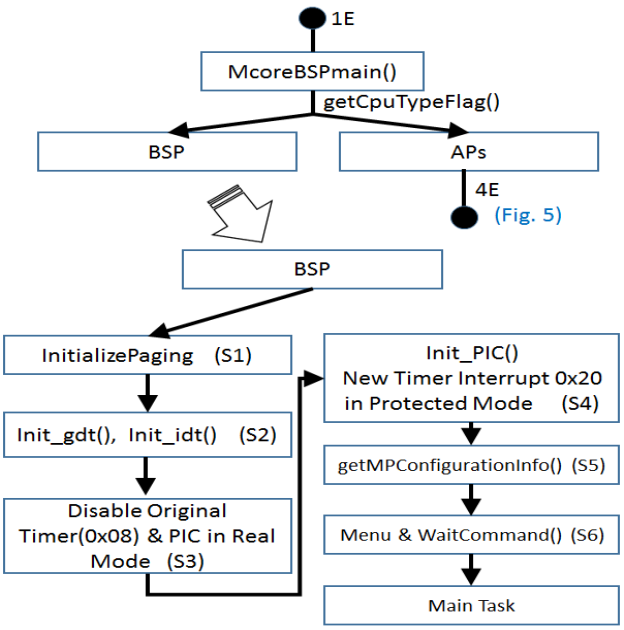

Figure 3. BSP Flow (1E)

is created and stored in advance in a stack to serve client requests. When a client's HTTP Get request arrives, an existing task from the task pool is popped and inserted into a circular list. When this task is complete, it is pushed back onto the stack for reuse. The 32-bit server can run over 6000 concurrent tasks on a single core Dell Optiplex 260 [15]. The code size is about 636 sectors, which is the entire AO needed for the application. The server is designed strictly as a Web server and only performs HTTP requests, which provides protection against conventional attacks that require an OS and other functionality:

A. BSP Flow

The Web server executable has two separate executables. The first is prcycle.exe (PEX), which has the boot, load and startup code. PEX loads and runs the second executable, test.exe (TEX), which is the Web server application. The display menu enables a user to load, run and debug programs in addition to multi-core options. The Optiplex 9010 uses a quadcore 64 bit processor model. These cores play different roles in the bare PC Web server. The first core is referred to as a boot strap processor (BSP); the other three cores are known as application processors (AP1, AP2, and AP3). Initially, BSP runs the boot code and loads the Web server application executable. The BSP is automatically configured to run by BIOS (basic in-put/output system) and APs are not enabled at boot time. In the boot loader code, we set a CPU flag (0x01) at location 0x7c00 (boot address) to identity BSP as a boot processor for the Web server system. When BSP recognizes its role, it resets the CPU flag (0x00) at location 0x7c00 before activating APs.

BSP begins execution at an entry point main(), which is same in the 32-bit Web server code. Figure 2 shows this entry point and also an extension point "1E." This extension point shows the additional code needed for migration (shown in Figure 3). The entry point in BSP starts with McoreBSPmain() function. This reads the CPU type flag (using getCpuTypeFlag()) as stored in the boot sector, and identifies itself as a BSP processor as described above. The BSP calls InitializePaging() to create and initialize paging data structures (S1). The 32-bit Web server used GDT, LDT and IDT (interrupt descriptor table) entries that were implemented in PEX using TASM code. These entries work only in real mode and the Web server code has interrupt gates to transfer between real and protected modes (real<>protected). Thus, TEX runs in protected mode and PEX runs in real mode. In order to reach hardware interfaces, it uses (real<>protected) mode cycle which serves as a bridge between these two modes.

IDT entries are used to address interrupt service routing and GDT entries are used to address the application's code, data and stack. In order to migrate the 32-bit code, we need to implement new GDT and IDT entries and eliminate LDT (new processors use protected mode GDT and IDT structures). Also, PEX is written in TASM (64 bit compilers do not support this assembler). Thus, new GDT and IDT entries were created, and initialized using init_gdt() and init_idt() functions (S2). The new entries GDT and IDT can be used in protected mode, which is required for migration. For example, 32-bit Web server code used a real mode timer interrupt at 0x08, whereas the new code uses 0x20 interrupt in protected mode. The PIC (programmable interrupt controller) chips have to be disabled (S3) and new APIC (advanced PIC) [13] (p.25) have to be enabled for migration (S4).

The next step in the migrati on process is to obtain multiprocessor (MP) configuration information [13] (p.37) (S5). This is implemented in getMPConfigurationInfo() function. This information gathering can be done in many ways [13] (p.38). The MP configuration information is used to manage APs and interrupts. The basic information stored in the MP configuration includes PIC mode, virtual wire mode via local APIC, virtual wire mode via I/O APIC, and I/O symmetric mode.

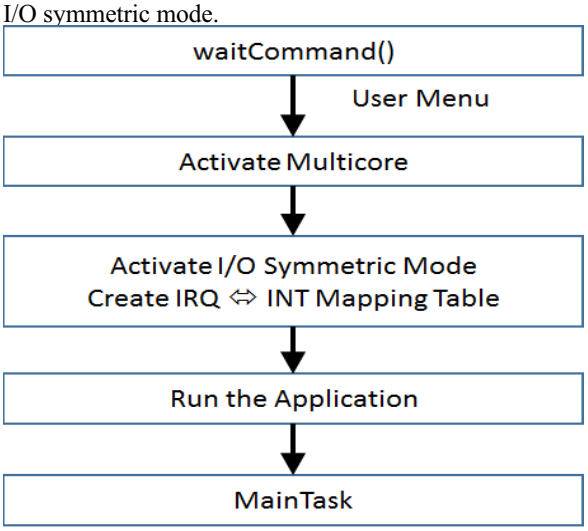

Figure 4. Wait Command Flow

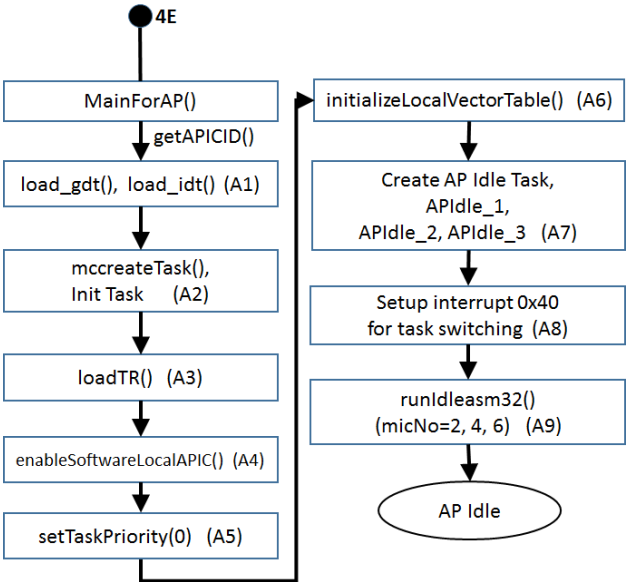

Figure 5. AP Flow (4E)

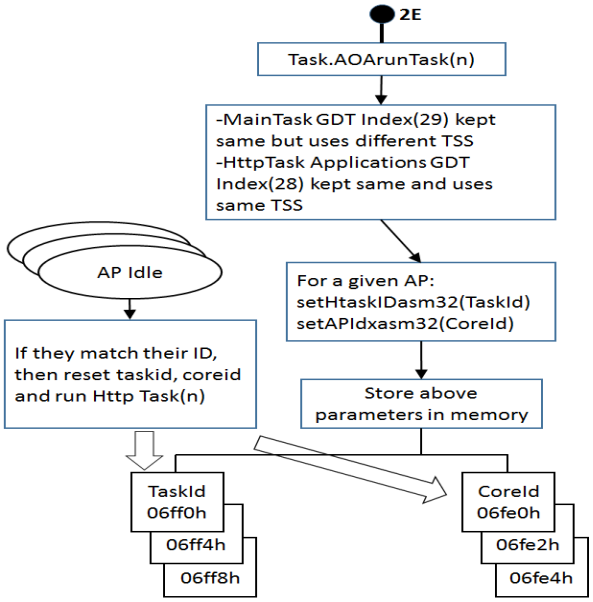

Figure 6. AP IDLE Task

The information is stored in Extended BIOS memory area in many places as it is accessed by a floating pointer structure. It is necessary to obtain the MP Floating Pointer Structure Address (e.g 0x000fda10, _MPFLOATINGPOINTER), which can be found in one of the three places in memory: (1) first KB of extended BIOS data area (EBDA); (2) within the last KB of system memory (639K-640K for 640K systems and 511K-512K for 512K systems); or (3) BIOS ROM address between 0f0000h and 0fffffh. As the information is "floating" in the above areas, we need to search those areas with a signature _MP_ to obtain the information. In the system used for migration, we found this address in the EBIOS memory area.

Using this address, we accessed the MP Configuration Table address (for example, the 0x000fd6d0, _MPCNFIGURATIONTABLEHEADER). The MP floating pointer structure in this system requires use of the MP configuration table instead of the default MP configuration, and virtual wire mode support instead of PIC mode. The first processor entry structure address can be obtained by adding 0x2c to the MP Configuration Table address (0x00fd6d0), i.e. 00fd6fc. The subsequent processor entries can be found by adding another 20 bytes for each entry. The processor parameters listed in the _PROCESSORENTRY are used to identify a given processor complex and configuration. The processor parameters obtained in this step are used to manage multi-cores in the system including interrupt vectors and control. The MP configuration structures are complex and suited for conventional OS-based systems as they use system calls and other libraries. Managing these structures in a bare PC, which uses the bare machine computing paradigm, is non-trivial and requires thorough understanding of the intricacies of 64-bit architecture and multi-core architecture to migrate 32-bit applications.

The BSP invokes waitCommand() function (S6), which provides a sequence of commands to run an application as shown in Figure 4, these options are provided by user Menu. Each processor has its own APIC to control interrupts. The interrupt requests come to an I/O APIC controller, which will be delegated to each processor APIC. Thus, to activate multi-core, the BSP performs two steps. The first step involves reading the MSR register using the "rdmsr" assembly instruction, ORing with the 0x800 value and writing back using the "wrmsr" assembly instruction. In the second step, it activates its own local APIC by changing the spurious interrupt vector register (0xfee00000 (base) + 0xf0 (offset)) with a value 0x100 [13] (p.77). At this point, interrupts are directed to only the BSP. In order to direct interrupts to other APs, we need to activate I/O symmetric mode. In symmetric mode, it is not possible to use virtual wire mode interrupt routing [13] (p.85).

The MP configuration table must be set up to do appropriate interrupt routing to APICs. This implementation is complex and depends on the APIC architecture and interrupt load balancing. In this system, timer interrupts are directed to APs and all other interrupts are processed by the BSP. Finally, when run option provided by the uer Menu is chosen, the BSP creates MTSK and starts executing that task. At this point, the BSP is in MTSK and all other APs are activated and running their individual idle tasks.

B. AP Flow

Once the interrupts are set properly, we can create an idle task which will run in the AP as a never ending loop in the processor (A7). We created three idle tasks (APIdle_1, APIdle_2, APIdle_3) with their own GDT indexes (i.e 0x208, 0x210, 0x218) and their corresponding GDT entries. We have also created one IDT entry for each AP: an interrupt vector 0x40 operates as a task gate to do task

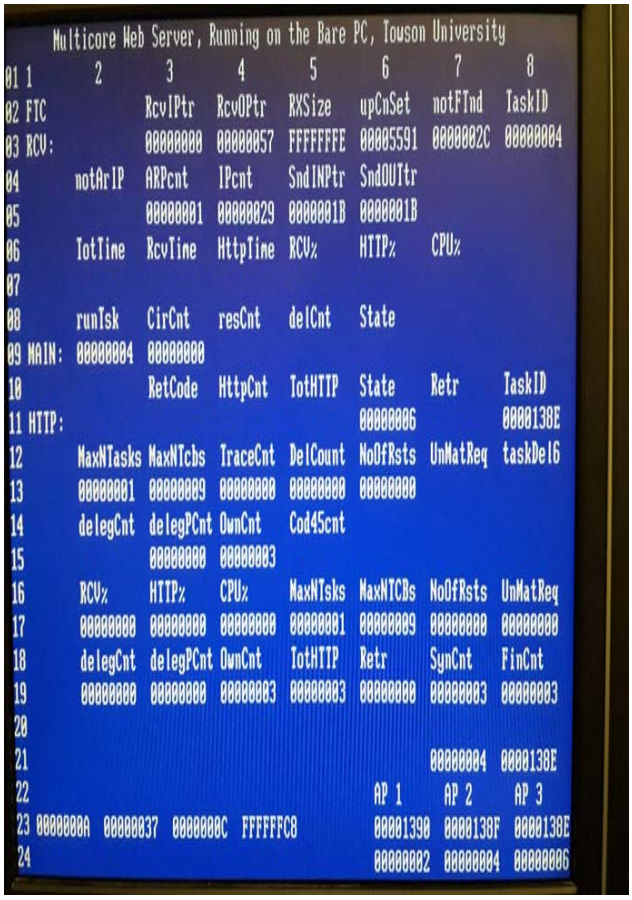

Figure 7. Multi-core Bare PC Display

switching from init task to idle task (A8).

Each AP will run its own idle task and monitor for any tasks assigned to it by the BSP. In order to go to idle task, we need an interrupt 0x40 to do task switching. The idle task will be running and waiting for any future tasks to be assigned by the BSP processor. The runIdleasm32() will enable the task switch from init to idle task by invoking interrupt 0x40. Figure 6 illustrates some details of creating the idle task in the AP.

C. Task Scheduling Flow

The This section describes the migratory code added for the Web server at point "2E" in Figure 2. When a Get request arrives from an HTTP client, the server pops an HTTP task (HTSK) from the task pool and inserts it into a circular list. Each request is an independent HTSK that handles data transfer and termination of the task. The connection part of the HTSK is done by RTSK. The MTSK and RTSK run on the BSP only. When there is only one core, all tasks are in the same processor.

When a HTSK is ready to be dispatched, it calls the function runTask(n) to run a task with task id n. In order to run task n, we have to do task switching from MTSK to HTSK. MTSK GDT index (in the 32-bit server) is 29. The new code uses the same GDT index with a new TSS. In the 32-bit server, HTSK has its own GDT index (28) and its own TSS. The new code uses the same GDT and TSS structures with no change. The task switch logic is not changed for migration. When APs are enabled, for a given AP, we set task id and core id in a shared memory location for each p

rocessor, where a core can access its own coreid. setHTaskIDasm32(taskid) and setAPIdx2asm32(coreid) are functions that will store the values in shared memory at 0x06fe0 and 0x06ff0 for core 1. Similarly, for core 2 and core 3, the values are stored in shared memory at different locations. As described before, APs are running in an APIdle loop and check for the match of their coreid in the shared memory. If the coreid matches, the core will invoke runTask(n) to run the HTSK. In a given AP, when the HTSK is complete, it returns back to the idle task. In the 32-bit Web server, there is only one processor to schedule MTSK, RTSK and HTSK. As shown in Figure 2, "3E" connection point illustrates the new configuration for migration. In this case, we have BSP, AP1, AP2 and AP3 processors connected to the circular list. These execution elements have to be load balanced to achieve high performance, which is not the focus of this paper.

V. FUNCTIONAL OPERATION AND DATA

After Results of performance studies using the 32-bit bare PC Web server in single mode and in split server mode are given in [3] and [15] respectively. This Web server is implemented using C/C++/MASM/NASM languages. The new code implemented for migration includes 1,595 lines of assembly and 9,366 lines of C/C++ code (with comments). Additional code will be needed for optimization and load balancing of cores in the future. The server is tested on an Optiplex 9010 with 4 core for functional operation. It is tested using an IE browser (Internet Explorer 11, but can be any other browser) and the http_load stress tool [16]. This section includes functional testing measures and data to validate the design and implementation of migration.

In the first migration attempt, no multi-core operations are used. The http_load stress tool is used to collect this data. This data shows the stress tool output for 600 seconds requesting 1000 requests per second. The important data to be considered is mean connect time (0.185272 milli-secs) and first response time (0.169373 milli-seconds) measured in this run. We next considered single core with paging. In this case, http_load was run for 1000 requests per second over a 10-minute period. The output shows the mean connect time is 0.193595 milli-secs and first response time is 0.145881 milli-secs. It can be seen that this is a little slower than in the first attempt in connect time, but response time is faster. We also run this with and without paging. There is not much of a difference in their performance as all page entries are populated for the given size of memory, and there were no page faults in the bare implementation.

VI. DISCUSSION

Our migration approach results in a unique bare machine Web server design that leverages the multi-core architecture. Many intricacies are involved in implementing the multi-

core architecture on a bare PC. The implementation details provided will be useful not only for bare PC systems but also for other application domains and platforms such as sensor networks and embedded systems, where there is no need for a conventional OS. Migration provides a basis for future research in innovative multi-core designs for bare machine applications.

The first attempt at migrating bare PC code illustrates the flexibility of this code and the ability to make it work with just a few changes on a 64-bit system when 64-bit/multi-core features are not used. Several novelties of bare PC applications are evident in the migration outcomes. For example, we saw that it is possible for an end user program to have complete control of applications, interrupts, GDT, IDT, MP configuration, and direct hardware interfaces. The multi-core code developed is completely independent of any other environment, application, or external software i.e. it only depends on the underlying multi-core processor and Intel architecture. To optimize performance it is also necessary to investigate synchronization and concurrency control mechanisms needed for bare PC applications running on multi-core. Future possibilities for research include migrating other bare PC applications to multi-core in a similar manner, and applying the split-server concept to multi-core by splitting protocols, applications and individual components.

VII. CONCLUSION

We presented a novel approach for migrating a bare PC Web server from a single processor system to a multi-core system. Our work also demonstrated a first step in migration that does not require any multi-core features to be implemented. We further showed how paging could be implemented in a basic bare PC system to make it compatible with the 64-bit architecture. In particular, we detailed how to run applications on multi-core processor systems and configure, control and manage their internal elements, including registers and control data needed for implementing a working system. This paper also discussed novel features of this approach and some possibilities for further research. The methodology used for migrating the 32-bit Web server will enable existing and future bare PC applications to be run on a multi-core 64-bit architecture. Moreover, the bare machine computing paradigm can be used to build simple, scalable, secure and high performance systems by exploiting the upcoming growth in multi-cores.

REFERENCES

[1] R. K. Karne, K. V. Jaganathan, N. Rosa, and T. Ahmed, "DOSC: dispersed operating system computing", 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005, pp. 55-61.

[2] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ applications on a bare PC", SNPD 2005, Proceedings of NPD 2005, 6th ACIS International Conference, IEEE, May 2005, pp. 50-55.

[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), 2009, pp. 480-485.

[5] A. Alexander, A. L.Wijesinha, R. Karne, "Implementing a VOIP SIP server and a user agent on a bare PC", 2nd International Conference on Future Computational Technologies and Applications (Future Computing), 2010.

[6] G. Khaksari, A. Wijesinha, R. Karne, L. He, and S. Girumala., "A peer-to-peer bare PC VoIP application", IEEE Consumer Communications and Networking Conference (CCNC) 2007.

[7] TinyOS, http://www.tinyos.net, accessed December 17, 2015.

[8] 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 (IPDPS), 2010.

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

[10] T. G. Brutch, "Migration to multicore: tools that can help", Login: The USENIX Magazine, Vol. 34, No. 5, October 2009.

[11] F. Nemati, J. Kraft, and T. Nolte, "Towards migrating legacy real-time systems to multi-core systems", IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2008, pp. 717-720.

[12] J. Schneider, M. Bohn, and R. Robger, "Migration of real-time automotive software to multicore systems: first steps towards an automated solution", 22nd Euromicro Conference on Real-Time Systems (ECRTS), 2010.

[13] MultiProcessor Specification Version 1.4, Intel, 1997. http://download.intel.com/design/archives/processors/pro/docs/24201606.pdf.

[14] Intel® 64 and IA-32 Architectures Software Developers Manual, Intel, 2015. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[15] B. Rawal, R. K. Karne, and A. L. Wijesinha, "Mini Web server clusters for HTTP request splitting", IEEE International Conference on High Performance Computing and Communications (HPCC), 2011, pp. 94-100.

[16] http_load – event loop based multiprocessing http test client, http://github.perusio.org/httpload, accessed December 17, 2015.

[17] Mihai Pricopi, Tulika Mitra, "Task Scheduling on Adaptive Multi-Core", IEEE Transactions on Computers, 2014, pp. 2590 - 2603.

[18] Shouzhen Gu, Qingfeng Zhuge, "Optimizing Task and Data Assignment on Multi-Core Systems with Multi-Port SPMs", IEEE Transactions on Parallel and Distributed Systems, 2015, pp. 2549 - 2560.