Transforming a Bare PC Application to Run on an ARM Device

Alexander Peter, Ramesh K. Karne, Alexander L. Wijesinha, Patrick Appiah-Kubi
Department of Computer & Information Sciences
Towson, MD 21252
apeter9@students.towson.edu, (rkarne, awijesinha, appiahkubi)@towson.edu

***Abstract*—Bare machine applications currently run on x86-based CPUs without any operating system or kernel support. Their low overhead makes them especially suited for mobile devices and pervasive computing. As an initial step towards running bare applications on mobile devices, we transform an x86-based bare PC graphics application to run on an ARM device. We first identify key differences between the x86 and ARM architectures relevant to the transformation. We then describe a methodology to transform the x86-based bare graphics application to run on the ARM architecture. We also present timing measurements when drawing graphics functions using the same bare application on an x86 bare PC, ARM development board, DOSBox emulator, and QEMU-VM simulator. This work provides insight into designing future bare machine applications that can run on a variety of mobile and pervasive devices with minimal code changes.**

Keywords-Bare Machine Computing (BMC); transforming applications; mobile devices; pervasive computing; ARM architecture.

I. INTRODUCTION

The ARM architecture is widely used in mobile devices including mobile phones and smartphones. We consider the problem of transforming a bare PC application that does not use any form of operating system (OS) or kernel to run on an ARM processor. Bare PC applications have low overhead since they run directly on the underlying hardware, which makes them a good match for low-power mobile devices. They are also suited for pervasive computing in view of their ability to run without any additional support or software. While a variety of bare PC applications have been developed previously, they run on devices with x86 CPUs. Transforming a bare PC application to run on the ARM architecture is a first step towards investigating their potential for mobile and pervasive computing.

Bare applications are self-contained and independent of any OS, kernel, or execution environment. They are written to self-manage the hardware assuming a CPU architecture based on x86 (the hardware API, tasking mechanisms, and drivers are therefore x86-specific). Transforming an x86-based bare PC application to run on the ARM architecture is non-trivial since there is no OS or kernel support of any kind to act as an intermediary on behalf of the application. By understanding how to write bare applications in a general manner, the transformation process can be simplified. It will then be possible to run the same bare application code (for example, a bare VoIP client, a bare lightweight Web browser, or a bare sensor application) on a variety of devices with minimal code changes.

In this paper, we solve the transformation problem from x86 to the ARM architecture for a particular application. Specifically, we transform a graphics application that runs on an x86-based bare PC so that it will run on a device with an ARM processor. The transformation is achieved by making minimal changes to the existing x86-based bare PC graphics application. We also measure the time to run the same bare graphics application on an x86-based PC, an ARM development board, DOSBox emulator, and the QEMU-VM simulator.

The rest of the paper is organized as follows. Section II provides an overview of bare machine computing and related work. Section III compares the principal differences between the x86 and ARM architectures from a bare application transformation viewpoint. Section IV discusses the transformation process in detail. Section V gives the timing measurements, and Section VI contains the conclusion.

II. BACKGROUND AND RELATED WORK

Computer applications are typically written in high level programming languages, which at compile and link time require OS calls in some form. These calls enable applications to access hardware resources at run time. In contrast, bare machine applications eliminate the OS and run directly on the hardware. Previously developed bare machine applications include Web servers [1]; email servers and clients [2], [3]; SIP servers [4]; applications secured via IPsec [5]; VoIP softphone clients [6]; and split protocol servers [7]. These applications are based on the bare machine computing (BMC) paradigm, previously called the dispersed operating system computing (DOSC) paradigm [8], which does not use any OS, kernel, or embedded system calls. Bare applications directly communicate with the hardware using their own bare interfaces to the hardware [9]. The BMC paradigm was used in [10] to develop a graphics application that runs on an x86 bare PC. This paper transforms the code of that bare PC graphics application so that it runs on an ARM device.

Subsequent to the proposal in [11] to eliminate OS abstractions, many studies have addressed the problem of reducing OS overhead for applications. Exokernel [12], OS-Kit [13] and Tiny OS [14] provide examples of lean kernels, OS components, or a small OS for sensors respectively. In contrast, the BMC paradigm allows an application programmer to have sole control of the application and its execution environment. In this paradigm, application programs communicate directly to the hardware via the above-mentioned hardware interfaces. These interfaces can, for example, enable program load, screen display, mouse and keyboard access, process management, and network and audio card control if needed by an application. Each application can thus carry only those specific interfaces that are needed for its execution. When the OS, kernel, or any form of intermediary system software is eliminated, resource management for an application is simplified. This results in much less overhead than for a conventional application as shown in [1], [2], [6].

In the areas of code transformation and translation, many innovative techniques have been proposed. For example, Intel x86 programs can be translated from binary code to ARM and

Alfa binaries with reasonable code densities and quality [15], [16]. The hardware abstraction layer concept is used to implement the Java virtual machine directly on hardware in an embedded system (as extensions of standard interpreters and hardware objects that interface directly with the JVM) [17]. In SoulPad [18], an auto-configuring OS with a suspended virtual machine is created on a small portable device at boot-time, giving users access to their personal environment and previously running computations. In BulkCompiler [19], a simple compiler layer is provided that works with ISA primitives and software algorithms to drive instruction-group formation, and to transform code to exploit groups. Our work differs from previous code transformation/translation approaches in that we are transforming the same bare code so that it can run on different CPU architectures.

III. ARCHITECTURE COMPARISON

We now examine the main differences between the x86 and ARM architectures that impact the transformation process. In an x86 bare application (such as a graphics application [10]), a bootable USB and PC BIOS are used to boot the PC. The boot process starts in real mode (address space 1MB) in an x86 system. In protected mode for x86, the address space is 4GB. There is no notion of real or protected mode in an ARM system. Instead, it has several mode types in system mode, and it also has a user mode. These modes are used by an OS to provide protection to applications and the system [20].

The user display in x86 is controlled by video memory (0x B8000 for text mode): a user places the data in memory and it is immediately displayed on the screen. In the ARM architecture, a low-level driver is needed to display information on the screen. In x86, mass storage can be provided by hard disk (HD), solid state disk (SDD), optical disk (CD/DVD) or USB. In ARM, mass storage is provided by SD (Secure Digital non-volatile memory card), NAND Flash, NOR Flash, and USB; however they are not similar to a PCI (peripheral component interconnect) bus in x86. In x86, Intel's Host Controller Hub (ICH) provides interfaces to many I/O devices. In ARM, each I/O device has a controller that is directly connected to the CPU and provides GPIO (general purpose I/O with 187 pins) to the programmer [21].

Table I summarizes the comparison. It should be noted that there are many other differences (and similarities) in the x86 and ARM architectures that are not considered here.

TABLE I. ARCHITECTURE COMPARISON

Characteristics	**x86**	**ARM**
Architecture	CISC	RISC
Development Environment	Visual Studio, C/C++	Eclipse CDT, GNU C/C++
Boot	BIOS	X-Loader, U-boot
Address Space	Real(20), Protected(32)	User(32), System(32)
Display	Video Memory	Driver
User Input	Keyboard/Mouse	Touch Screen
Mass Storage	HD, SDD, USB	SD, USB

C/C++ compilers and linkers are available for both x86 and ARM devices. As expected, the assembly language and CPU instructions are different in these devices. Thus, any assembly level code/instructions used in x86 must be re-written for ARM processor with respect to its differences in attributes as shown in Table 1.

IV. TRANSFORMATION PROCESS

A high-level methodology for transforming a small graphics application from x86 to ARM is shown in Fig. 1. This approach can be adapted to develop a general transformation methodology in the future for other types of bare machine applications.

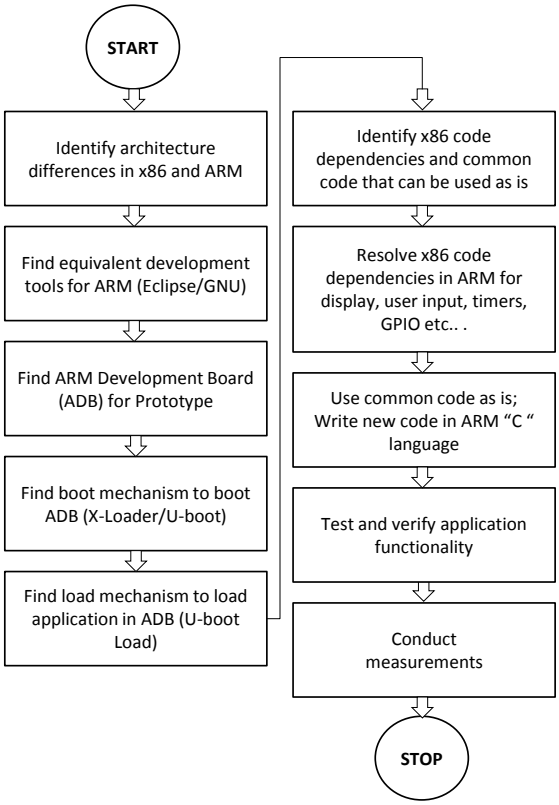

Figure 1. Transformation process.

Except for the boot and load process, the graphics application code transformed from x86 is bare code that does not depend on any OS or kernel. The following subsections provide details of the application transformation process.

A. *Graphics Application on x86*

The bare PC graphics application [10] is designed to perform graphics functions, such as drawing geometric figures with fractal primitives, displaying text characters, pixels, lines, circles, and bitmap images, and handling attributes such as color. The basic graphics functions implemented using standard C and Intel assembly language include screen access, screen framework, font/symbols, image/video, shapes, attributes, image transformation and bit operations for performance as described in [10].

The number of lines of C code for the application is 615 (executable lines) and the number of lines of assembly code is 766 (including comments). The number of lines of assembly code for all hardware interfaces in a bare PC is 1969 (including comments). We only use about 10% of the

hardware interfaces for the graphics application. The boot and load code are not counted in this measurement.

B. Transformation to ARM

The overall flow for transforming the bare x86-based application to an ARM application was shown in Fig. 1. The key requirement is that the application should have the same functionality when running on an x86 bare PC and an ARM device.

B.1 Development Environment

Eclipse IDE was used as the development environment for writing the ARM code on a laptop or desktop system. The code for x86 was generated using batch files and Visual Studio.

B.2 Development Board

In order to have full control over the development process and to also make the system bare, an ARM development board (OK6410 from Samsung) was used. This board comes with built-in (conventional) systems including Android, Win-CE and Embedded Linux. It has 256 MB main memory, 2 GB mass storage, and 667 MHz clock speed. A SanDisk 2GB SD card was used for secondary storage [21].

B.3 Boot and Load

We used the U-boot tool, which is a universal boot loader [22] that is easy to use; it also provides free source code. The interactive commands provided by the tool can be used to boot, load, and execute programs. We only use minimal U-boot functionality including CPU Register/Stack setup (before call to an application), setup UART console, setup clock/PLL setting, and memory initialization. Although the boot/load process could also be made bare (similar to what is done in x86 bare application development), this has not been done for the bare graphics application. After the executable is created using the Eclipse IDE, it is loaded using U-boot's "loadb" command. Usually, an application is loaded at a specified address (e.g. 0x800).

B.4 Execution

After the executable is loaded into memory, it is executed by invoking U-boot's "go" command. The "go" command will load this specific start address in the PC register and start executing from this address [9]. In an x86 bare PC application, boot, load, and execute are all part of the application object (AO), which is controlled by the programmer [23].

B.5 x86 Code Dependencies

The dependencies specific to x86 that were found in the bare PC graphics application code are due to data types, memory addresses, type casting, pointers, device addresses and device interfaces. For example, video memory is used to display graphics, whereas a bare driver is used to write to the screen in ARM. It would be possible to reduce such dependencies if the original code is written with the intention of transforming it to run on a different architecture.

Fig. 2 shows code analysis that illustrates the transformation process. In this example, 38% of the code is reused from x86, 34% is newly written, and 28% is modified. The modifications are due to the dependencies identified above.

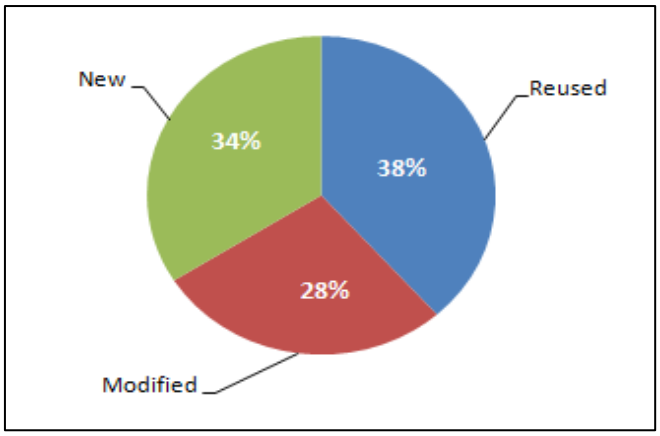

Figure 2. Code transformation x86 to ARM

The application in x86 uses its own boot, load, and execute written in assembly, which consists of about 1969 lines (including comments); the equivalent in ARM is U-boot. The assembly code used in bare (video memory access and controls) is about 766 lines (including comments), which is similar to the new code written for ARM. This code is 288 lines written in C. The total number of lines of C code in x86 is 615 compared to 580 lines for the ARM bare application. As expected, ARM code is smaller in size and appears to be more robust than the code in x86. In general, ARM is more suited for this type of bare machine application than x86.

B.6 Implement x86 Dependencies

As noted above, there are many x86-specific dependencies in the bare graphics application code. However, since most of these are trivial, except device interfaces in x86 versus ARM, we only discuss the latter. In effect, we will illustrate the implementation differences when drawing a pixel on the screen, which serves as a building block for accomplishing most of the graphics API.

The bare PC graphics implementation in x86 is shown in Fig. 3. The graphics application requires setup modes for controlling the graphics card. This is done by simply using BIOS interrupts. Once the graphics mode and parameters are initialized, subsequent graphics operations use direct hardware interfaces to store graphics data in video memory. The bare PC does not have privileged and user modes, so all operations are done in user mode only. Since BIOS interrupts only work in real mode, all graphics control operations use an interrupt gate mechanism to go from protected mode to real mode. This switch is transparent to the AO programmer since it is encapsulated in the direct hardware interfaces used by bare PC applications.

An AO can also use software interrupts to reach certain hardware facilities in the bare PC system as shown in Fig. 3. As bare PC applications do not use any graphics drivers, they rely on the direct hardware interfaces thus avoiding the need for any intermediary software in the system. We also use some shared memory in a bare PC application to facilitate direct communication between an AO and the underlying hardware. For example, a timer interrupt will update shared memory in

real mode and the timer value can be accessed by an AO using a direct shared memory interface.

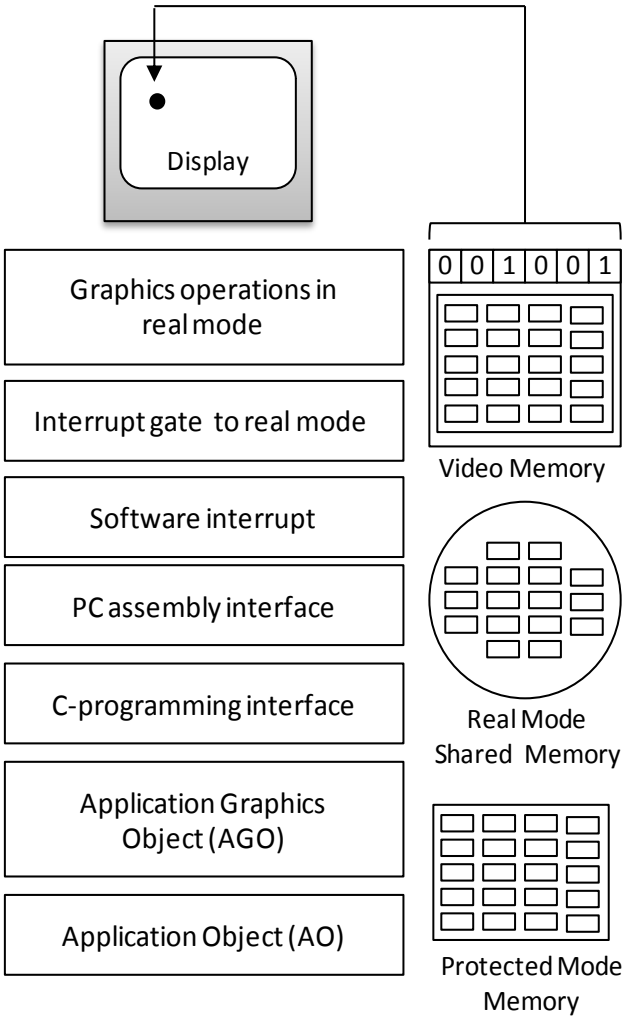

Figure 3. x86 bare graphics implementation.

B.7 Implement ARM Dependencies

The bare graphics implementation on ARM is shown in Fig. 4. Fig. 3 and Fig. 4 taken together illustrate the similarities of the AO and AGO (Application Graphics Object) implementations in the BMC paradigm. At startup, the whole screen is blank and the LCD backlight is off. Since the display driver is written in C and it is not using any external or third party code or libraries, all necessary registers are defined and the macros are setup for controlling the lines through GPIO (General Purpose Input/Output). First, the application executes the bare display driver. Via registers and macros, it initializes the vertical and horizontal cycles with precise timing requirements as specified on the LCD Controller data sheet and turns on the LCD backlight [24]. Second, the application initializes the Frame Buffer (FB), which is a location in memory to store the display data. Third, the bare application calls the draw_pixel() function with three parameters: the y, x coordinates and color of a given pixel. Fourth, the bare graphics interface modifies the first four bytes in the FB with data according to the given color. Fifth, the application draws whole FB frame on the screen. It goes in a loop, reading all data from FB and generating the necessary signals through GPIO for CLK (Clock), VSYNC (Vertical Synchorization), HSYNC (Horizontial Synchronization) and DATA read from FB [21], [24].

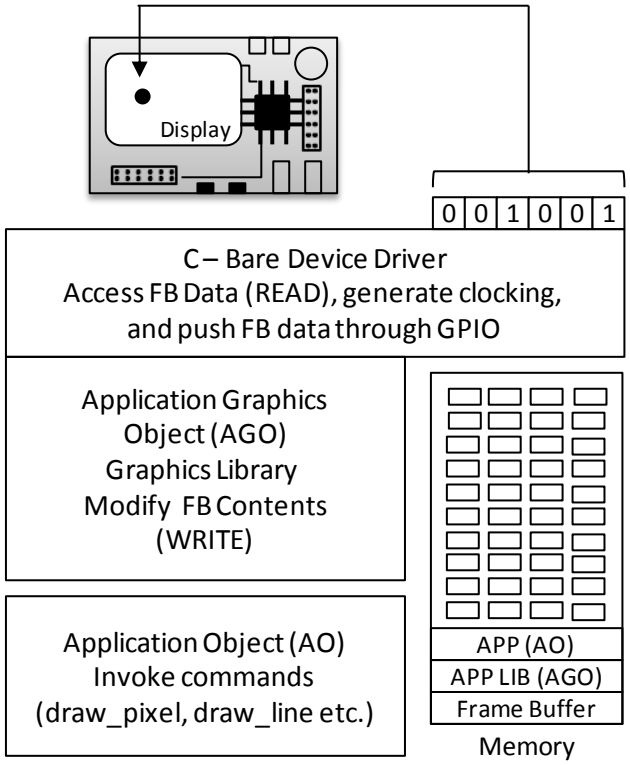

Figure 4. ARM bare graphics implementation.

V. TESTING AND MEASUREMENTS

The testing environment consists of an x86 bare PC, bare ARM development board, DOSBox and QEMU-VM simulator. For x86, the testing was conducted on a Dell Latitude D620 laptop with a standard VGA graphics card and VESA enabled BIOS on VGA Mode 13, with 320-by-200 pixel resolution in 256-Colors, and 1.83 GHz clock speed. For ARM, the testing was conducted on an ARM development board: Samsung S3C6410/OK6410 micro controller with 677 MHz, and 4.3'' WXCAT43-TG3 TFT-LCD panel [21], [24]. For the DOSBox emulator on Windows XP, the x86 graphics application code was written in Turbo C. This code uses BIOS calls for graphics. The bare PC graphics application was also run using the QEMU-VM simulator. Here, the same application code as for x86 is run on the simulator, which runs on Microsoft Windows XP.

Fig. 5 shows the testing/execution environment for the timing measurements given in this paper. The left laptop shows the display of images for a bare PC application. The right laptop shows the same images on a QEMU-VM simulator running on Windows. The ARM development board is shown with wires illustrating the connections. The bottom three screens show lines, circles, and images captured from the ARM display. It can be seen that these images look the same in all testing/execution environments.

For timing measurements, four graphics functions were used: draw_pixel(), draw_line(), draw_circle(), and draw_bitmap(). In an x86 bare PC, each pixel requires three

bytes: the first two bytes indicate the location/screen size, and the third byte has the color information. These three bytes are formed in video memory directly for the desired number of pixels. Since video memory is linear, the location is calculated by multiplying the y coordinate by the total screen width, and adding the x coordinate. The draw_pixel() function forms the 3 bytes data in video memory. When drawing a large number of pixels, we used random coordinates and colors. In the ARM implementation, we used the same draw_pixel() interface. However, since video memory is not accessible in the ARM development board, we put the pixel data in main memory and pushed it into the LCD. This process is done completely in software instead of using a hardware display controller. The code written for the bare ARM device to control the LCD includes GPIO initialization, clocking, vertical sync, horizontal sync, and data enable. It is independent of any OS or environment.

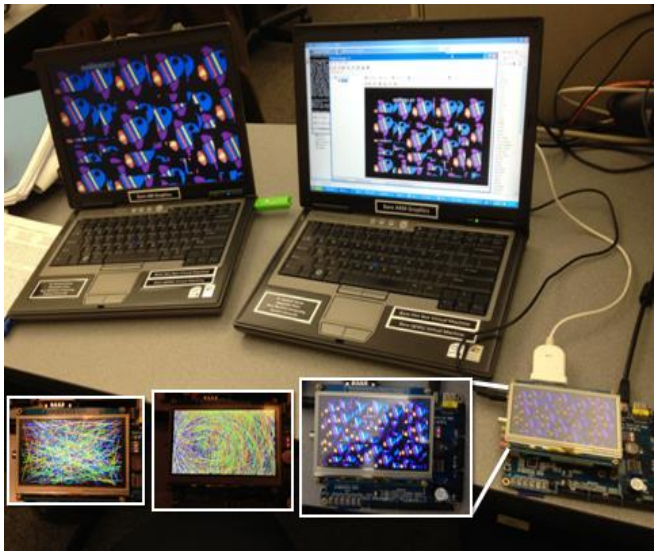

Figure 5. Executing and testing environments.

The timing results below only demonstrate the basic performance of the same x86 bare graphics application when it is transformed with a minimum of changes to run on ARM. A more detailed study is needed to account for differences in clock speed, instruction set and graphics controller implementation.

Fig. 6 shows the time to draw 500,000 to 75,000,000 pixels on x86, ARM, DOSBox, and QEMU-VM. It can be seen that the x86 bare PC has better timings than for the other systems as expected since it eliminates all OS overhead. Since the bare ARM development board uses a software display driver and its clock rate is 2.74 times slower than the x86 PC, it runs the slowest. The DOSBox is efficient since it uses BIOS calls and the hardware display controller. When drawing pixels, DOSBox takes less time than QEMU-VM.

Figs. 7-9 show the time to draw 50,000 to 3,000,000 lines, 50,000 to 400,000 circles, and 1000 to 26,000 images respectively. The timing difference between the bare PC and DOSBox are very small when drawing lines and circles, with a very small advantage for DOSBox over the bare PC in the latter case. Also, QEMU-VM performs worse than ARM for circles, but better than DOSBox and ARM for images (DOSBox shows a drop in performance for images). More studies are needed to determine the reasons for these variations in performance.

The individual performance gains for the ARM development board and the x86 bare PC are shown in Fig. 10. The performance gains for drawing pixels, lines circles, and images range from 1–11, 3.5–60, 8–11 and 0.2–27.8 respectively. These results indicate that the average gain for the x86 bare PC ranges from 6.5–35. Notice that the gain for drawing lines is much higher than for the other graphics. The likely reason is that drawing lines requires more calculations for computing the slope, which is faster in the x86 bare PC due to its clock speed and complex instructions for long operations.

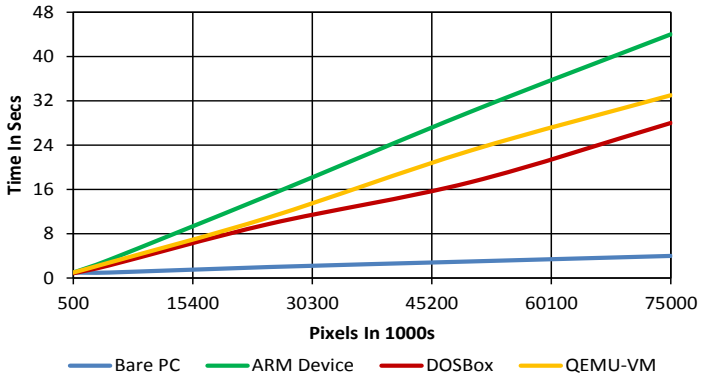

Figure 6. Time to display pixels.

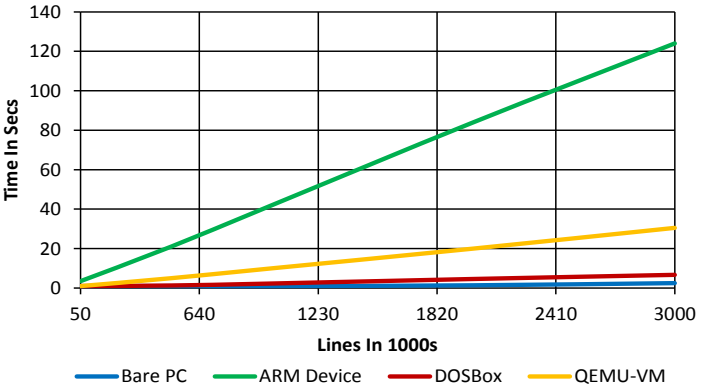

Figure 7. Time to display lines.

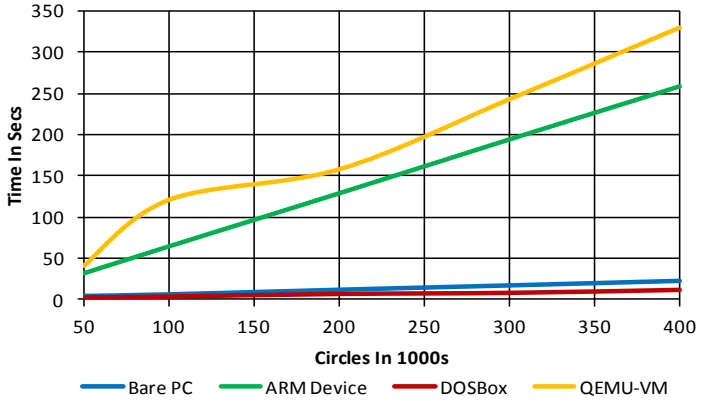

Figure 8. Time to display circles.

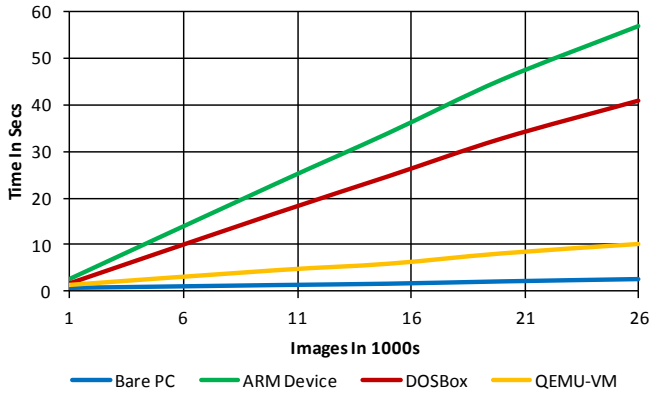

Figure 9. Time to display images.

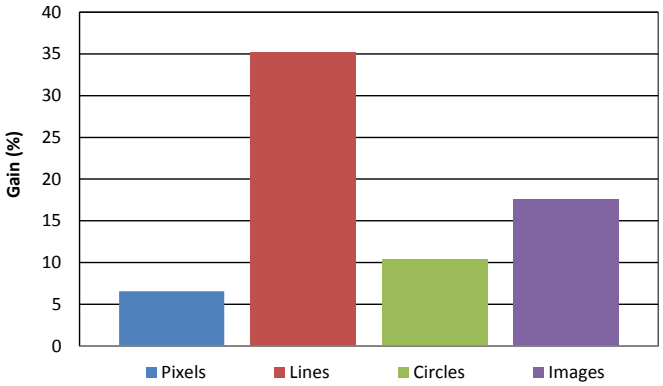

Figure 10. x86 bare PC gain over a bare ARM device.

VI. CONCLUSION

Bare machine applications that run on the hardware without any operating system or kernel support have many characteristics that are suited for mobile computing. As a first step towards investigating this potential, we transformed an x86-based bare PC graphics application to run on the popular ARM architecture. We identified differences in the x86 and ARM architecture relevant to the transformation, and outlined a transformation methodology. The graphics application code can be classified into three categories: code that can be used as is, code that is modified, and new code that is written for ARM. The transformed x86 application code running on ARM has no external dependencies other than on U-boot for boot and load. The transformed bare application was tested using an ARM development board. We also did basic timing studies to measure performance of the bare application when running on the x86 and ARM architectures. We also determined timings when running the application on a DOSBox emulator and QEMU-VM simulator. Although performance on the x86 bare PC is in general better than on ARM, DOSBox, and QEMU-VM, more detailed studies are needed to evaluate the transformed bare application code on ARM. The methodology presented in this paper can serve as a basis to develop a general approach to transform other x86 bare applications for use in mobile devices, and for pervasive computing.

REFERENCES

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

[2] P. Appiah-Kubi, 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 (AHPCN), pp. 521-526, 2010.

[3] G. Ford, R. 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.

[4] A. Alexander, A. L. Wijesinha, and R. Karne, "A Study of Bare PC SIP Server Performance," The Fifth International Conference on Systems and Networks Communications (ICSNC), pp. 392-397, 2010.

[5] N. Kazemi, A. L. Wijesinha, and R. Karne, "Evaluation of IPsec Overhead for VoIP using a Bare PC," 2nd International Conference on Computer Engineering and Technology (ICCET), pp. 586-589, 2010.

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

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

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

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

[10] A. Peter, R. K. Karne, A. L. Wijesinha and P. Appiah-Kubi, "The Design and Implementation of Bare PC Graphics," 7th International Multi-Conference on Computing in the Global Information Technology (ICCGI) pp. 315-320, 2012.

[11] R. Engler and M.F. Kaashoek, "Exterminate all Operating System Abstractions," Fifth Workshop on Hot Topics in Operating Systems, p. 78, 1995.

[12] D. Engler, "The Exokernel Operating System Architecture," Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Ph.D. Thesis, 1998.

[13] The OS Kit Project, School of Computing, University of Utah, http://www.cs.utah.edu/flux/oskit. Last Accessed Aug. 2012.

[14] Tiny OS, Tiny OS Open Technology Alliance, University of California, Berkeley, CA, 2004, http://www.tinyos.net/. Last Accessed Aug. 2012.

[15] 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.

[16] Y-S Hwang, T-Y Lin, and R-G Chang, "DisIRer: Converting a Retargetable Compiler into a Multiplatform Binary Translator," ACM Transactions on Architecture and Code Optimization (TACO), vol. 7 (no. 4), pp. 18-33, 2010.

[17] M. Schoeberl, S. Korsholm, T. Kalibera, and A.P. Ravn, "A Hardware Abstraction Layer in Java," ACM Transactions on Embedded Computing Systems (TECS), vol. 10 (no. 4), pp.1-40, 2011.

[18] R. Cáceres, C. Carter, C. Narayanaswami and M. Raghunath, "Reincarnating PCs with Portable SoulPads," IBM T.J. Watson Research Center, New York.

[19] W. Ahn, S. Qi, M. Nicolaides and J. Torrellas, "BulkCompiler: High-Performance Sequential Consistency through Cooperative Compiler and Hardware Support," MICRO'09, pp. 133-144, 2009.

[20] W. Hohl. "ARM Assembly Language. Fundamentals and Techniques," Florida: CRC Press, 2009.

[21] Samsung Electronics Inc. South Korea. User's Manual S3C6410X/OK6410 RISC Microprocessor Rev 1.20.

[22] L. Edwards. "Embedded System Design on a Shoestring," Boston: Newnes, 2003.

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

[24] WANXIN IMAGE Inc. Taiwan. 4.3'' TFT-LCD with Touch Panel Module Model #: WXCAT43-TG3#001 Product Specification Document.