A PORTABLE, CROSS-PLATFORM EMULATION SYSTEM

John Casey [1]

[1] School of Information Technology, Deakin University,
221 Burwood Hwy, Melbourne, VIC 3125, Australia

ABSTRACT: Traditionally, emulation systems have been unnecessarily written and compiled for specific computer architectures. Therefore, emulations cannot easily be ported to other computing platforms. This paper proposes a portable emulation system, where the dependencies on particular host architectures are minimised by developing the emulation runtime, in the portable environment Java. Using this system architecture, the emulation can be executed on any computer platform that has a Java runtime component. Using this framework, a pilot system has been developed that emulates a subset of the 8086 processor using interpretation. The relative performance of the emulation is then evaluated by comparing the benchmark results with those attained by the portable emulation and the natively compiled 80x86 Bochs emulation, which uses a similar interpretive mechanism to the pilot system. Additionally, the system is compared to the 8086, 80486 and Pentium 3 processors. The benchmark programs are processor bound test sequences of 8086 instructions, and are comprised of specific groupings of similarly formatted instructions. The performance of the pilot system is better than the 8086 but not as good as the 80486, Pentium 3 or Bochs emulator. The performance differences between the pilot system and Bochs highlight the performance trade-offs that exist between portability and raw performance.

INTRODUCTION

Computer emulation is a software migration technique that allows legacy software systems to execute as they were designed on newer hardware and software platforms. Emulation software provides translation mechanisms using various techniques to reproduce the logic and design of the system(s) the software was originally developed for.

Thus, the process of emulation development is naturally an extremely complex and time consuming task. Currently, a number of emulation systems have begun to use processor definition files and driver based emulation frameworks. These emulation architectures allow the core emulation component to be re-used in the development of multiple emulation systems for various host environments.

In this paper, a simpler architecture is proposed, where the entire emulation system is written specifically for a virtualised, portable environment such as Sun's Java. Using this approach, the complexities involved in the development of a portable emulation can be reduced to the equivalent of developing a standalone emulation. Such a system can be used on any computer platform, where a Java Virtual Machine is available. It is expected that such a portable emulation will be able to decode and execute instructions many times faster than the original 8086, when running on modern hardware. However, the performance differences between the portable emulation and the Bochs emulation are expected to highlight the performance, portability trade-off that exists in such portable environments.

This paper is organised as follows. Section 2 examines the related work. Section 3 deals with the architecture and development of the portable emulation proposed in this paper. In Section 4 the performance of this emulation system is evaluated, and compared with contemporary emulation and hardware systems. Finally, section 5 concludes the paper and discusses the further work.

RELATED WORK

This section is divided into two parts. The first part discusses current emulation processes and technology, whilst the second part examines portability issues in emulation. In the first part an overview is given of the various emulation processes, as well as an indication of their relative performance and compatibility. The second part solely focuses on issues of portability, reusability, and their use in emulation.

Emulation Processes

Interpretation is the baseline emulation technique, and mirrors the fetch, decode, and execute loop that processors use to execute programs. In emulations, this process typically emulates one instruction at a time. Unfortunately, interpretive emulation mechanisms carry a high overhead that can slow emulation speed. The interpretive algorithm is the simplest emulation mechanism, and is able to faithfully replicate exotic programming constructs such as polymorphic code. Interpretation is used in several emulation systems, such as the PC emulation Bochs [Law99] and Apple's 68000 Macintosh emulation [App94].

In comparison, static binary translation systems recompile the machine code of a program to that of a target system. Once an executable program has been translated, it can be run as a native program on the target computer. Translated executables can run at near full speed without any interaction with the translation program. However, binary translation cannot correctly translate polymorphic code. Consequently, many translators work in conjunction with interpreters and use the interpretive method as a fall back approach. A number of popular emulation systems use this technique such as Digital's Freeport express [Dig95], and FX!32 [HH97] systems among others.

Dynamic binary translation is an extension to static binary translation, and uses a just-in-time approach to generate a cache of native machine code instructions. These translated instructions can be called directly by a host systems processor, without having to repeatedly re-interpret emulated instructions. The dynamic nature of these translation processes allows emulation software to identify all sections of program code. Presently, dynamic translation is becoming the dominant emulation mechanism in many emulation environments, such as Apple's re-written 68k Macintosh emulator [App95], ARDI's Executor [Hos95], IBM's DAISY environment [EA97] and notably in virtual machine environments such as Sun's Java[Sun01].

Portable Emulations

Emulations are complicated systems to design, develop and implement. As such, an in depth and intimate knowledge of the hardware and software systems used in both the source and target computer systems is required.

The engineering process of emulations typically locks the system into a single source and target system. In recent years this problem has been overcome somewhat, by re-writing emulation systems in portable languages such as ANSI C and C++. This allows an emulation system to be rebuilt for any target platform, where an appropriate compiler is available. Unfortunately, while a target system can be manually adjusted and retargeted to suit multiple platforms, the source or emulated system is hard coded as program statements and instructions. Additionally, such emulations may not be as hardware independent as they could be, due to limitations of the language and hardware specific optimisations.
Consequently, this has lead to the development of emulation systems which rely on processor and system specification files, which define the format, syntax and behaviour of hardware and software functions. [CS97] have developed SSL, which is an instruction specification encoding language. In their research, they define the syntax and semantics of the instructions in a procedural manner, which specifies the interactions of an instruction with the registers and address space of a processor.

A similar process is used in Syn68k [Hos95], a portable dynamic binary translation system that emulates the 680x0 family of processors and is used in Executor, the Macintosh emulator. The core of the Syn68k emulation is based around an instruction specification system called Syngen and this is used at compile time to generate the system's emulation and translation functions. At compile time,

Syngen optimises the code generated from the specification for different host environments, based on the architectural details of particular processors and software systems. However, the focus of Syngen is optimisation rather than portability.

Using a similar automated process to Syngen, the University of Queensland's dynamic binary translator (UQDBT) [UC00] provides a core emulation framework that is supplemented with code generated from semantic specific files. In contrast to the performance focus of Syn68k, the UQDBT system centres on retargeting the core emulation, so it can support various source and target computer systems. However, UQDBT cannot be easily retargeted to new platforms so easily. The system building process requires new system definition files to be created, and for the entire system to be recompiled and retested as well. Therefore, the source code of the UQDBT emulation is portable between different platforms, whilst the compiled executables are not.

Other emulation toolkits, notably the open source MAME (Multiple Arcade Machine Emulator) [Mam96] and MESS (Multiple Emulation Super System) [Mes98] projects use a modular, driver based architecture. Using this structured approach, the functionality of various processor and I/O device emulation modules can be re-used through the use of simple driver programs. These driver programs glue processor and I/O device modules together to emulate an entire computer system. The system architectures of these systems are similar to retargetable systems in that the basic core system can potentially support numerous emulations. However, MAME and MESS differ in that they support emulations using pre-compiled code libraries, rather than the specification file and code generation techniques of Syn68k and UQDBT.

The retargetable and driver based emulation frameworks examined here provide a fast method for developing processor efficient emulation systems. Unfortunately, these system frameworks depend on pre-existing system definition files and processor emulation libraries. Consequently, using these architectures, the development of emulations can be quite complex.

To appreciate the complexity of the development process, consider the following examples. In retargetable emulation frameworks, the most common architectural elements of a processor must be supported as well as the more exotic features as well. Therefore, the core functionality of these emulations can be complicated with functions that may not be necessary for the emulation of common processor designs. Conversely, the core emulation may be missing necessary emulation functions for particular computer architectures, which means more functionality will have to be developed and added to the system.

As with retargeting emulations, the development of driver based systems can also be complicated. These systems require a library of pre-written processor and I/O device emulations, and these will take a long time to develop if they do not exist already. Moreover, to support the emulation of new computer architectures, extra processor and I/O device emulations will have to be developed and added to the library system.

The research documented in the remainder of this paper examines the performance characteristics of such a portable emulation system and compares it with the performance of a natively compiled emulation, and several comparable hardware systems.

DEVELOPMENT PROCESSES

This section outlines the architecture, steps and processes involved in the development of a portable emulation system. Initially, the scope, goals and requirements of the system architecture will be examined. This will be followed by an examination of the different system components that comprise the system architecture.

For the purposes of this study, the emulation development focuses on a subset of the entire emulation process. Consequently, the emulation system has concentrated on the core components of an emulation system. These core components include the processor module, memory system and register set. As such, the emulation does not attempt to replicate a computer's input or output systems. The Intel 8086 architecture has been selected as the target of this emulation, as contemporary systems are still able to execute 8086 programs.

As this system is a pilot study, interpretation has been selected as the primary method of program execution. As stated in the literature, the interpretive algorithm is very inefficient. However, the simplicity and accuracy of interpretation compensates for the performance problems associated with the method. Moreover, it is unlikely a dynamic binary translation system could be implemented easily within the Java environment as there is no simple way to generate useable byte-code at runtime.

Finally, the instruction set of the emulation has been limited, so that only common instructions and addressing modes were emulated. Consequently, the processor emulation does not represent a full emulation, rather a test-bed system suitable for testing purposes only. Therefore, the primary purpose of the emulation is to generate program execution times, which can be used to compare the performance of portable emulation to that of the corresponding hardware systems.

Candidate instructions and addressing modes were selected for emulation on the basis of how often they would be used in general 8086 programs. Using this approach approximately 60 percent of the 8086 instruction set was emulated.

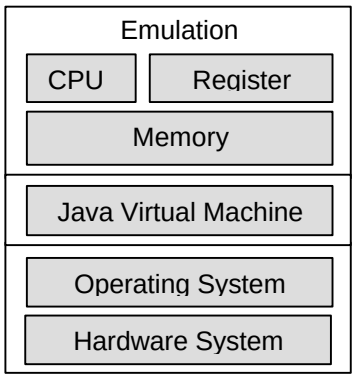

Figure 1

Figure 1 illustrates the layered architecture of the emulation system. As shown, the Java virtual machine acts as an abstraction layer that allows the emulator to run on any computer platform.

The CPU module is the main component of the emulator. It repeatedly interprets and executes instructions using a switch decode loop, and uses function calls to execute decode instructions. These functions may interact with the systems memory or register objects.

The memory component comprises an integer array, which spans the 8086's 1 MB address space. Decode/execute functions can read or write values to memory simply by indexing the integer array. Simple array indexing is fine for reading and writing single byte values. However, read() and write() routines have been written for addressing word values, which have to be byte-swapped for the little endian memory format of the 8086.

The system registers simply use integer variables to represent the word addressable registers of the 8086 and these are updated using assignment statements. However, the data registers of the 8086 are both byte and word addressable. Therefore, with these registers a separate class structure has been developed with methods that allow the register to be accessed as a word, or as a low or high byte value. Bitwise masking operations are used to perform these addressing functions due to the lack of a union structure in Java.

PERFORMANCE ANALYSIS

To evaluate the proposed emulation architecture, a series of benchmarks have been developed to examine the instruction execution performance of the system. The benchmarks consist of simple processor bound programs that group the machine code instructions that have similar formatting and function. This allows the performance characteristics of different instruction groupings (such as stack, memory transfer, arithmetic) to be analysed.

Performance is measured, by recording the program execution time of each benchmark program as it is executed on the various test systems. These test systems comprise the Intel Pentium 3/700 MHz, Intel 80486 DX2/66 MHz, Intel 8086/8 MHz, and Bochs, a native 80x86 emulator. The Bochs emulator uses a similar interpretive architecture to the portable emulation proposed in this paper. The Pentium 3/700 MHz system is used to execute the portable emulation, as well as the Bochs emulator.

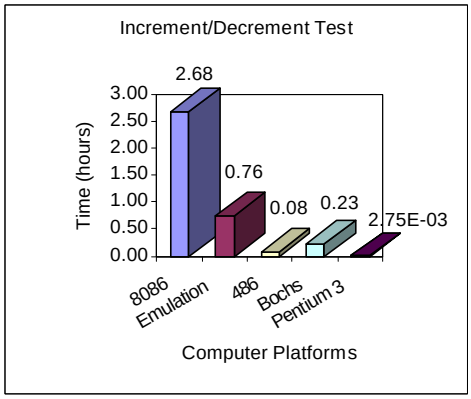

Figure 2 Increment/Decrement Test

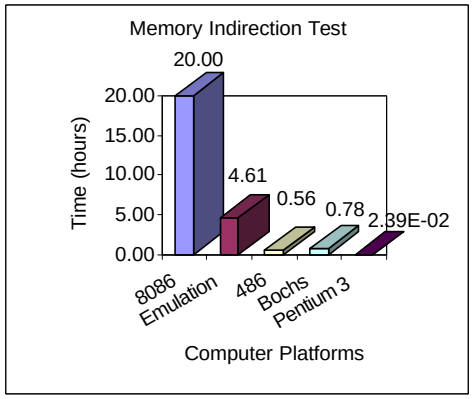

Figure 3 Memory Indirection Test

For each experiment, the core benchmarking code that tests the speed of the various instruction types is executed 10,000 times for each test program. These programs are structured so that they will make use of the various addressing formats of the 8086 processor, such as the immediate, register, direct, indirect, 8 and 16 bit addressing modes. The results generated after 10,000 iterations are presented in figures 2 and 3. The y-axis represents the time duration taken to process a benchmark, whilst the x-axis represents the different computer platforms examined in the investigation.

In figure 2, the results of the increment/decrement series of benchmarks are presented. Computationally, these are the simplest testing procedures. From the results, we can see the 8086 processor has the longest execution time at 2.68 hours, and that the portable emulation is faster at 0.76 hours. However, relative to the 80486 and Pentium 3 processors and Bochs emulation we can see the portable emulation is slower.

Figure 3 presents the results of the indirect memory addressing benchmarks, which execute a number of complex addressing instructions. These instructions are primarily used to perform array indexing and copying operations. The complicated encoding format of the array indexing instructions has resulted in a dramatic increase in emulation overheads. The results of this benchmark show that the 8086 again has the longest program duration at 20 hours, followed by the emulation at 4.86 and Bochs at 0.78. Again, in this test case the emulation is dramatically slower than the 486 and Pentium 3 processors.

The remaining benchmark results show very similar trends to the preceding graphs. These are summarised in the table below.

Table 1 Benchmark Results Program Execution Time (Hours)/10,000 iterations

Benchmarks	8086	Emulation	Bochs	486	Pentium 3
Increment/Decrement	2.68	0.76	0.23	0.08	0.00
Multiply/Divide	11.84	1.35	0.21	0.23	0.01
Direct Memory	6.96	1.94	0.36	0.18	0.02
Addition/Subtraction	7.43	2.38	0.30	0.13	0.01
Indirect Memory	20.00	4.61	0.78	0.56	0.02
Stack Operations	5.97	1.18	0.30	0.16	0.01

Overall, the results emphasize that emulations are highly dependent on the complexity of an instruction's encoding. For example, the results of the increment/decrement benchmark test are quite favourable when compared to the performance of the Bochs emulator and the 486 processor.

In particular, the close results of the portable emulation with Bochs in the increment/decrement test highlights the performance gap between Java programs and natively compiled C/C++ programs. Additionally, it can be seen that instructions that make use of the 8086 mod/rm byte to address different register and memory combinations incur a higher performance penalty than the other addressing modes.

In general, all the results of the testing phase present a similar trend, where the results show the emulation outperforming the original 8086 processor. However, on average the results indicate the emulation is many times slower than the Pentium 3 and 80486 processors. Nevertheless, across the benchmarks the results can be compared favourably between the portable emulation and the natively compiled Bochs system.

CONCLUSION AND FURTHER WORK

The research presented in this paper addresses the feasibility of a portable emulation system. The results indicate that on the current generation of computers, portable emulation is possible, but at a considerable cost in execution performance.

From the results, we can conclude that at the moment portable emulation is really only feasible for vintage computer processors. This means it may be possible to implement a full system emulation that also replicates an entire system's input and output devices as well.

The iterative structure of the benchmark tests highlight the inefficiencies associated with the interpretive emulation approach. Throughout the emulation process, the temporal naivety of the interpretive algorithm is evident. Moreover, interpretive emulation systems do not take advantage of iterative programming constructs, where a group of instructions are repeatedly called.

Consequently the interpretive approach interprets sequences of iterative instructions repeatedly, unaware that they have previously been decoded. In effect, the iterative nature of the benchmarks highlights the major drawbacks and weaknesses of the interpretive emulation approach. This interpretive overhead coupled with the additional overheads of virtual machine architecture tends to make portable emulation very slow.

Clearly, a method of caching the previously interpreted and translated instructions for later use is required. Such a system would comprise a binary translation component, which is capable of translating native 8086 instructions into Java's byte code instruction format. On subsequent executions of pre-translated code blocks, the emulation would be able to simply recall the translated procedures. This would substantially improve the performance of the emulation, as the overheads associated with interpretation could be minimised.

Unfortunately, the restrictions of the Java environment make it difficult to translate sequences of 8086 machine code into Java byte-code. The nature and design of the Java language hinders the development of programs which require pools of memory to be set aside for the generation of self modifying code. Nevertheless, it is believed that the core translation component of such a system could be developed if programmed at the byte-code level. Such a system could avoid the restrictions of the Java language and exploit the untapped features of the Java virtual machine. However, such modifications at the byte code level could have unforeseen consequences for byte-code cache coherency, portability and problems with the optimisation routines of the Java virtual machine.

REFERENCES

[App94] Apple 1994, *Inside Macintosh: PowerPC System Software*, Addison Wesley, Massachusetts.

[App95] Apple 1995, *The DR emulator*, Available: [http://developer.apple.com/technotes/pt/pt_39.html] (2/3/2002).

[CS97] Cifuentes, C. & Sendall, S. 1997, 'Specifying the Semantics of Machine Instructions', in *International Workshop on Program Comprehension*, IEEE-CS Press, Italy, pp. 126-133.

[Dig95] Digital 1995, *Alphamigration Tools: Freeport Express*, Available: [http://www.support.compaq.com/amt/freeport/index.html].

[EA97] Ebcioglu, K. & Altman, E. R. 1997, 'DAISY: Dynamic Compilation for 100% Architectural Compatibility', in *Proc.ISCA 24*, ACM Press, New York, pp. 26-37.

[HH97] Hookway, R. J. & Herdeg, M. A. 1997, 'Digital FX!32 Combining Emulation and Binary Translation', *Digital Technical Journal,* vol. 9, no. 1, pp. 3-12.

[Hos95] Hostetter, M. 1995, *Syn68k ARDI's dynamically compiling 68LC040 emulator*, Available: [http://www.ardi.com/SynPaper/index.html] (10/4/2002).

[Law99] Lawton, K. 1999, *Bochs*, Available: [http://bochs.sourceforge.net/] (18/6/2003).

[Mam96] MAME 1996, *MAME - The official Multiple Arcade Machine Emulator site*, Available: [http://www.mame.net/] (18/6/2003).

[Mes98] MESS 1998, *The Official MESS Home Page*, Available: [http://www.mess.org] (18/6/2003).

[Sun01] Sun 2001, *The Java Hotspot Virtual Machine*, Available: [http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf] (16/10/2002).

[UC00] Ung, D. & Cifuentes, C. 2000, 'Machine Adaptable Dynamic Binary Translation', *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, pp. 30-40. Retrieved: Jan 2000, from