Preserving Host Independent Emulation Speed

Peter Jakubčo[1], Liberios Vokorokos[1]

[1] Faculty of Electrical Engineering and Informatics, Technical university of Košice, Letná 9, 042 00 Košice, Slovakia
{peter.jakubco, liberios.vokorokos}@tuke.sk

Abstract. Degree of emulator's ability to imitate emulated machine is defined by its accuracy. EmuStudio platform, used mainly for educational purposes, tries to be an accurate emulator. The most widely implemented accuracy level is preservation of emulated computer speed. Modification of host machine hardware affects the emulation speed and therefore it is necessary to guard if the emulation speed match the requirements.

Keywords: emulation, sequential algorithm, emulation accuracy

1 Introduction – emulator accuracy

The emuStudio emulation platform is software developed at Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical university of Košice. The design aim is to develop an emulator able to emulate arbitrary computer configurations that are based on von Neumann architecture conception [1].

The main characteristic of each emulator is its accuracy that defines the scope how the emulator behaves like real machine. There exist several accuracy levels. The most important are:

a) accuracy of data and states internal representation,
b) and accuracy of timing (preserving instruction cycles time).

The effort of accuracy achievement on the one level can decrease accuracy achievement on the other level. For example, skipping some specific details of emulated hardware can decrease the emulation quality itself. On the other hand, replacement of certain real hardware elements by proper abstraction, often improves the emulation quality. Therefore it is essential to decide what accuracy level is needed (depending on emulator's nature), and how can be achieved.

2 Basic emulator's structure

Main activity of running emulation is to perform exact sequence of well-defined steps repetitively. It meets with the work of real computer, because the activity of a real computer (mainly instruction execution) can be divided into repetitious steps sequence.

The emulator's work is represented by an algorithm (Fig. 1), where a scheduler is implemented. The scheduler allots host processor’s time to particular emulated components, including emulated processor, in the exactly defined order.

```
running = true;
(numberOfCycles, maxTime) = initTimeSyn();
while (running) {
    running = executeCPU(numberOfCycles);
    ints = generateInterrupts();
    emulateDevices(ints, maxTime);
    (numberOfCycles, maxTime) = timeSyn();
}
```

Fig. 1: Basic emulator algorithm (sequential).

The scheduler allots certain time interval to each emulated component in the same sequence. In the allotted time the component can perform any activity. The time interval is estimated, and depends on required emulation speed.

When the time interval expires, time synchronization must be performed. It is not guaranteed whether the components will be running in the exact boundaries of the allotted time. Sometimes their emulation cannot be stopped right after time expiration – that depends on the nature of emulated component.

Usually the time interval is estimated for single emulation iteration for all components together. At the end of the iteration the time synchronization is performed. Time interval in the algorithm in Fig. 1 is divided into two parts:

- numberOfCycles – maximum number of processor machine cycles that the processor can execute for single emulation iteration;
- maxTime – maximum common time interval in what all devices can work within single emulation iteration.

Besides of central processing unit (CPU) and devices emulation, external interrupts are generated, causing the setting of corresponding state variables of CPU emulator, along with the need of certain conditions satisfaction. The interrupts are generated by peripheral devices. Their handling is realized effectively only when the interrupt generating is located outside the processor execution (outside of the executeCPU procedure).

3 Emulation of central processing unit

The core of every single computer emulator is processor, or CPU (Central Processing Unit).

The emuStudio platform does not limit the programmer in the use of emulation technique. However, till the present time each implemented processor uses only interpretation. The skeleton of the processor emulation algorithm, using interpretation as emulation technique, has the form shown in the Fig. 2.

Devices emulation algorithm is using two methods – readIO and writeIO. Their task is to identify a device connected to processor's port (defined in the operand of an input-output instruction) and call the input/output of the device by a way that is defined in the communication model. Quick response of such two methods is very important.

```
cycles = 0;
while (cycles < numberOfCycles) {
    opcode = fetch_opcode(PC);
    PC = PC + 1;
    switch (opcode) {
        case MOV:
           ...
           cycles = cycles + MOV_CYCLES;
           break;
        ...
        case IN:
            port = fetch_operand(PC);
            PC = PC + 1;
            A  = readIO(port);
            cycles = cycles + IN_CYCLES;
            break;
        case OUT:
            port = fetch_operand(PC);
            PC = PC + 1;
            writeIO(port, A);
            cycles = cycles + OUT_CYCLES;
            break;
        ...
}
```

Fig.2: Algorithm of CPU interpretation (`executeCPU` procedure)

4 Keeping the required emulation speed

One requirement of a reliable emulation (and the satisfaction of timing accuracy) is to stabilize the speed of the emulated computer and to synchronize it with the required speed. Mostly the demand is to reach the real speed of a real computer that is going to be emulated, but this is not a must.

Relative speed of computer emulation S_e where a program P is running is defined as:

$$S_e = \frac{f_{CPU}^e + t_{dev}^e}{t_P^e}, \qquad (1)$$

where:

- f_{CPU}^e– average frequency of the emulated processor that is executing the program P (s^{-1})
- t_{dev}^e – average throughput of other emulated components (s^{-1})
- t_P^e – throughput of emulated program P – number of significant changes (treatment of inputs and outputs) in the program for single time unit (s^{-1})

Relative speed of the real computer S is defined analogically:

$$S = \frac{f_{CPU} + t_{dev}}{t_P}, \qquad (2)$$

where:

- f_{CPU} – average frequency of the real processor executing the program P (s^{-1})
- t_{dev} – average throughput of the other real components (s^{-1})
- t_P – throughput of the program P – number of significant changes (treatment of inputs and outputs) in the program in the single time unit (s^{-1})

If the speed S is known, then the speed represents *required* speed ratio.
The requirement of speed synchronization is fulfilled, if the following equation holds:

$$S_e = S \qquad (3)$$

However such state is optimal and mostly it cannot be sustained for longer time period. If:

$$S_e < S, \qquad (4)$$

then the emulation is slowed-down, if compared with the required speed. Emulation speeding-up is not always possible to perform (it is always easier decreasing the performance as increasing). If:

$$S_e > S, \qquad (5)$$

the emulation is speeded-up compared to demanded speed, and it is necessary to slow down it in a certain way.

The principle of speed synchronization with required speed (in a case of *(4)* and *(5)*) is to regulate the speed.

Slowed-down emulation (condition *(4)*) can be regulated (increased) by the limitation of devices emulation quality (e.g. by reduction of the number of frames per second (fps) of graphic card, or by reducing the sound quality, etc.). The aim is to reduce emulation time of the other devices and to save a time for processor emulation.

Speeded-up emulation (condition *(5)*), in the contrast, can be easily decreased by reducing the number of instructions execution by the emulated processor. The "missed time" (freed processor's emulation time) then can be utilized either by increasing the emulation quality of other components, or by waiting in the idleness.
EmuStudio platform solves only condition *(5)*, because of the following reasons:

1. increasing the emulation speed is difficult and sometimes impossible,
2. in the present time the platform implements only older 8-bit computers, so the performance of present host machines is sufficient for their emulation.

Basic condition for successful speed regulation of the emulated processor is acquaintance of the number of machine cycles (CPI, Cycles Per Instruction) [4], needed for each instruction execution. Sometimes the number cannot be determined easily, because it depends on the instruction complexity.

Next important information is that for each period of a processor clock frequency, *n* machine cycles of an instruction are performed:

$$1\, T_{clock} \mathrel{\widehat{=}} n \cdot CPI \qquad (6)$$

Let's suppose that $n = 1$. Then a time needed for *i*th instruction realization equals to:

$$T_i = T_{clock} \cdot CPI_i \qquad (7)$$

It would be possible to regulate the speed of each individual instruction particularly, by what the most precise regulation could be achieved (sampling and regulation frequency would be high), but, on the other hand, the burden of regulation itself could slow down the emulation rapidly.

Therefore, in the emuStudio platform, the speed is regulated only after several instructions execution that is performed without the regulation. It is supposed that the host computer is much more powerful than required performance of emulated processor. Such approach, the "letting out" of non-regulated group of instructions, tries to overrun required speed. On the one hand, this saves some time, but on the other hand, the instructions are performed faster than they should perform. Anyway, in the overrunned time the regulation is performed. This time can be also useful in the other way than for regulation: it balances missed time lost in peripheral devices emulation, when they had run longer time than they should have.

Processor emulation algorithm with time synchronization is shown in Fig. 3. The algorithm uses following operations:

- now_time – finds out actual time (e.g. in *ms*)
- wait – waits a certain time given as parameter (e.g. in *ms*)

The algorithm begins by the definition of a fixed time interval (timeSlice), for which the group of non-regulated instructions should be performed. This value is designated empirically.

Then it is computed how many machine cycles would fit into the time interval. This parameter is called numberOfCycles, and has been used also in the previous algorithms.

```
running = true;
while (running) {
    startTime = now_time();
    cycles = 0;
    while ((cycles < numberOfCycles)) {
        opcode = fetch_opcode(PC);
        PC = PC + 1;
        i_cycles = execute(opcode);
        cycles = cycles + i_cycles;
    }
    // time synchronization
    endTime = now_time();
    timeLen = endTime - startTime;
    if (timeLen < timeSlice) {
        // time correction =
        // emulation is too fast
        wait(timeSlice - timeLen);
    }
}
```

Fig. 3: Algorithm of processor emulation with the speed synchronization

Final computation of the parameter comes from a certain computations sequence. The goal is to achieve equality *(3)*, therefore:

$$\frac{S_e}{S} = 1 \tag{8}$$

$$\frac{t_P \cdot (f_{CPU}^e + t_{dev}^e)}{t_P^e \cdot (f_{CPU} + t_{dev})} = 1 \tag{9}$$

and so:

$$\frac{t_P}{t_P^e} = \frac{f_{CPU} + t_{dev}}{f_{CPU}^e + t_{dev}^e} \tag{10}$$

Let's define speed ratio of the running program as:

$$P_R = \frac{t_P^e}{t_P} \tag{11}$$

Then if the emulated program is faster than it would be on the real computer, $P_R > 1$ holds. If it is slower, $P_R < 1$ holds. If the programs run at the same speed (what is the goal), $P_R = 1$ holds.

The emuStudio emulation platform takes into account both time for instructions execution and the time for devices emulation, in the single processor emulation iteration cycle. The devices emulation is realized immediately after input-output instruction execution. Therefore throughput of devices can be ignored, or added to emulated processor frequency:

$$f^e = f_{CPU}^e + t_{dev}^e \tag{12}$$

$$f = f_{CPU} + t_{dev} \tag{13}$$

From now, there are considered only processor frequencies. Then the following holds:

$$P_R = \frac{f^e}{f} \tag{14}$$

$$f^e = P_R \cdot f \tag{15}$$

$$\frac{1}{T^e} = \frac{P_R}{T} \tag{16}$$

$$T^e = \frac{T}{P_R} \tag{17}$$

where T^e is a period of emulated CPU, by which a single machine cycle of an instruction is performed. If we introduce computed numberOfCycles parameter into *(17)*, we get:

$$T^e \cdot numberOfCycles = \frac{T \cdot numberOfCycles}{P_R} \tag{18}$$

Then a single batch of instructions on real computer, that is emulated, takes:

$$timeSlice = T \cdot numberOfCycles \tag{19}$$

While emulating target computer on a host computer that runs at independent speed, the value of timeSlice parameter depends on host computer parameters.

The timeSlice parameter is designated empirically. Too low value can lead to significant emulation slow-down due to speed regulation. In the contrast, too big value would make the emulation "pulsation". It means that after each performed emulation iteration, the emulation would freeze in the idleness for significantly long time. There exists an empiric formula for timeSlice parameter computation:

$$timeSlice = 10 \cdot T^e \tag{20}$$

By expanding *(18)*:

$$T^e \cdot numberOfCycles = \frac{timeSlice}{P_R} \tag{21}$$

$$numberOfCycles = timeSlice \cdot \frac{f^e}{P_R} \tag{22}$$

Parameters f^e (required emulation frequency, or emulated processor frequency) and P_R(program speed ratio) are given. For speed preservation it must hold $P_R = 1$ and therefore final formula is:

$$numberOfCycles = timeSlice \cdot f^e \tag{23}$$

Seeing that for every frequency period a single machine cycle is performed, the parameter depends on required frequency, too.

Before and after each non-regulated instructions group emulation execution the starting time startTime and final time endTime are measured out.

By the difference of these times, we get the duration of the emulation iteration (timeLen). If it is compared with the timeSlice parameter, we find out how the real processor frequency is synchronized with required frequency (or speed):

$$timeLen = endTime - startTime \tag{24}$$

$$timeLen = timeSlice \tag{25}$$

$$timeLen > timeSlice \tag{26}$$

$$timeLen < timeSlice \tag{27}$$

If the emulation is speeded up (real processor frequency is bigger than required), there inequality *(27)* holds, and the algorithm in Fig. 3 solves that problem by waiting in the idleness (by the wait operation).

5 Conclusion

Emulation speed synchronization plays an important role. The correctness of many programs execution depends on exact timing of instruction cycles, executed on the processor. There exist two reasons for the effort of the achievement of the exact emulation timing.

At first, programmers are trying to utilize all processor performance. Older computers' programmers had to take into account how and when they could access to the devices (and perform wait loops). The goal was to minimize the wait loops duration. Certain device could be accessed only in an exact moment (e.g. video memory and video registers were accessible only if the picture was not drawing to the screen right now) and then it had been possible to compute how long time was needed for waiting.

Second, there exist several levels of emulator accuracy, and for the emulation of various hardware can be given various requirements. The more accuracy is required, the more consideration has to be taken into account for speed synchronization.

Acknowledgments. This work was supported by the Slovak Research and Development Agency under the contract No. APVV 0073-07 and VEGA grant project No.1/0026/10 "Modeling and simulation of security attacks in distributed computing environments and networks".

References

1. VON NEUMANN J. 1945. *First Draft of a Report on the EDVAC.* [online] Available from: <http://www.virtualtravelog.net/entries/2003-08-TheFirstDraft.pdf>.
2. JAKUBČO P., ŠIMOŇÁK S. 2009. *emuStudio – a plugin based emulation platform.* In: Journal of Information, Control and Management Systems, ISSN 1336-1716, 2009, vol.7, no.1, pp. K33–46
3. JELŠINA M. 2002. *Architectures of computer systems: principles, structural organization, function (In Slovak).* Košice: Elfa, 2002, 2. issue. 567p. ISBN 80-89066-40-2
4. VOKOROKOS, L. at all.: 2008. *Digital Computer Principles (in Slovak).* Elfa, s.r.o., 322p. ISBN 978-80-8086-075-2
5. JAKUBČO P, DOMITER M. 2010. *Standardization of computer emulation*, SAMI 2010 Proceedings, 8th International Symposium on Applied Machine Intelligence and Informatics, Herľany, Slovensko, 28.-30.1.2010, b.m., 2010, IEEE Catalog Number CFP0908E-CDR, pp. 147-151, ISBN 978-1-4244-3802-0
6. Arjan Tijms. 2000. "Binary translation: Classification of emulators" [online] Available from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.75.807&rep=rep1&type=pdf>.
7. BARRIO, V. M. 2001. *Study of the techniques for emulation programming*, Computer Science Engenieering, Facultat d'informàtica de Barcelona , Universitat Politècnica de Catalunya. [online]. Available from: <http://www.scribd.com/doc/94546/Study-of-the-techniques-for-emulation-programming-by-Victor-Moya-del-Barrio>.