A Fast Emulator for ARM-based Embedded Systems

Nariman Eskandari, Hatef Madani, Armin Ahmadzadeh, Mohsen Mahmoudi Aznaveh, Saeid Gorgin[*]

School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran.
HPC@ipm.ir

***Abstract*—This paper presents a high-performance implementation for an Intel 8080 emulator on a Raspberry Pi device. The problem was defined as a software contest in MEMOCODE 2014 and this implementation took the second place in this contest. We deployed several optimization techniques and employed best programming practices to increase the performance of the naïve reference implementation. Improving data structure usage and modifying function calls are the techniques that resulted in higher performance of this implementation. Our implementation has about 2.5 times speedup over the reference code of the contest.**

Keywords—8080 emulator, Raspberry Pi, Interpreter

I. INTRODUCTION

Hardware emulators have many applications in modern world from simulating unmanufactured ICs to game consoles. An emulator is software that takes the binary code of a platform and returns the equivalent binary of another platform that is actually running the code. Emulators are also used for reusing of obsolete platforms. In order to design an emulator, we must have in-depth knowledge of target machine. Using this information, every part of a machine can be emulated and systems' components can work in tandem.

To emulate a machine, its state must be stored and state transitions have to be correctly computed. Since the focus of this paper is on CPUs, the state of the CPU must be stored and transitions over CPU's state machine are calculated in an ARM-based hardware. CPU's state machine includes the content of RAM and registers, such as program counter register, stack pointer register and flag register.

There are two main approaches for emulating a CPU: using a CPU interpreter or employing dynamic recompilation [1]. CPU interpreter emulation consists of a loop in which a machine instruction is executed in a single iteration. The overall flow of the interpreter program is depicted in figure 1. Dynamic recompilation, also called binary translation, maps an operation from the source hardware into the actual host hardware [2]. In this work, we select the CPU interpreter approach, while the input program of the emulator is fixed. This approach is suitable for fixed programs.

```
While (executed_cycles<cycles_to_execute)
{
        opcode=memory[PC++];
        instruction=decode(opcode);
        execute(instruction);
}
```

Fig. 1. Main loop of a CPU interpreter [1]

[*]S. Gorgin is also affiliated with Iranian Research Organization for Science and Technology (IROST), Tehran, Iran.

According to CPU interpreter approach, there are three steps in instruction cycle: fetch, decode and execution. These steps are similar to the steps that a real CPU performs to execute an instruction. First, the opcode of an operation from the target machine's binary is fetched. Then, opcode is decoded and investigated to apply execution effect into machine's state stored in the RAM.

Flow of program is not just dependent on the instruction of a program. In fact, interrupts and branches have an important role in controlling the flow. Interrupts are generated by the hardware, like keystrokes, or from the program itself. In an emulator, the synchronization of the program that is influenced by interrupts must be handled by the programmer. The loop condition of figure 1 deals with this issue.

The rest of this paper is organized as follows: section II describes the contest problem and hardware specifications with its constraints. Section III introduces our proposed solution and section IV presents the experimental results using different inputs. Finally, we present conclusion in section V.

II. PROBLEM DESCRIPTION

This year's software contest is about emulator design. In an emulator program, a machine is emulated by another machine [3]. In this paper, source and destination (or host) machines refer to the emulated and emulator platforms, respectively. State of source machine is stored in RAM of destination machine that includes registers and memory of source machine. According to problem definition, the emulator must emulate Intel's 8080 processor, an 8-bit processor with 8 KB instruction memory and 2KB RAM. The destination machine is a Raspberry Pi, a 32-bit processor with 512 MB RAM. The aim of emulation is to launch an old nostalgic game (Space Invaders) and no other program does need to be emulated. The reference emulator can be found in [4].

The Raspberry Pi [5] is a palm sized computer that came into market with the purpose of having a low-budget single-board computer with a full operating system. It was manufactured to inspire children to use computers all over the world. However, it also gained a huge attraction from academia. Raspberry Pi is a general purpose device and it has ARM-11 core CPU. ARM-11 is a single core processor and does not support parallelism.

Raspberry Pi has also a GPU processor with 24 GFLOPs computing power. As the emulator does not require SIMD capability, GPU capabilities cannot be utilized in this problem.

Space Invaders game is the target program that must be run by the emulator. Space Invaders is an all-time classic arcade game from the 70s and is used for performance evaluation in this contest. Tomohiro Nishikado developed this game in 1978. This game was a pioneer in modern game industry. A screenshot of this game is shown in figure 2.

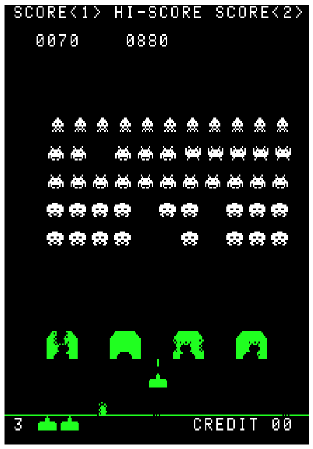

Fig. 2. Space Invaders screenshot

Space Invaders is a two-dimensional shooter game in which the player controls a laser beam by moving it across the bottom of the screen and firing at descending aliens. The player can defeat aliens by shooting them with laser cannon. The emulator runs this game in two modes on the destination platform: Play mode and Replay mode.

In Play mode, interrupts generated by players are gathered and saved in a specific file. Then in Replay mode, they are used as real-time inputs to the emulator in order to eliminate human player's decision-making delays.

The aim of the contest is to accelerate the emulator on the destination machine. Only software optimization is allowed to improve the performance of the emulation.

III. The Proposed Method

We have deployed several optimization techniques to accelerate running time of the emulator. Some of the techniques that were the most important ones are described in this section. Using macros and optimizing the data structure helped us to improve the performance of the emulator but threaded emulator [6] and system level optimization were not fruitful.

A. *Using macros instead of functions with high ratio calls*

Compilers convert function calls into jump instructions, which reduce the pace of the program because of divergences that flush the processor's instruction pipeline. Therefore, decreasing function calls yields some increase in program speed. We profiled the program, specified the function calls ratio, and converted the most called functions into macros. Figure 3 presents one of the functions (Figure 3(a)) that were changed to a macro (Figure 3(b)). Totally, we have changed 5 functions of the reference implementation to their equivalent macros.

```
//original function
Static void
ArithFlagsA(State8080*state,uint16_t
res)
{
     state->cc.cy=(res>0xff);
     state->cc.z=((res&0xff)==0);
     state->cc.s=(0x80==(res&0x80));
     state->cc.p=parity(res&0xff,8);
}
```

(a)

```
#define ArithFlagsA(state,res)\
  (state->cc.cy) = (res > 0xff);\
  (state->cc.z) = ((res&0xff) == 0);\
  (state->cc.s) = (0x80 == (res &
0x80));\
  (state->cc.p) = parity(res&0xff);
```

(b)

Fig. 3. (a) shows the selected frequent function from naïve code and (b) shows the same function after recoding to macro

B. *Data structure optimizatoin*

The other approach we used concerns the data structure to fully utilize the destination platform capabilities. Our investigation showed that some of the arrays and structures could benefit more from a special data structure that can be accessed faster. Usually, memory related instructions take many cycles to perform, as a result of memory latency, so it is important to reduce the number of such instructions efficiently to improve performance. Intel's 8080 processor has 8-bit registers but there are 32-bit registers available in destination hardware. Using a 32-bit register for an 8-bit register deteriorates performance in two ways: 1) it wastes memory bandwidth and 2) it forces us to use extra operation to mask the extra data. We improved the data structure that is shown in Figure 4.

One of the challenges we faced in this platform was how to manage both 8-bit and 16-bit data fetches. Program counter and stack pointer of the source machine are 16-bit registers. We handled them using union data structure in C.

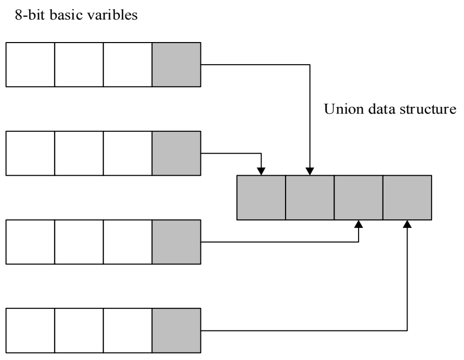

Fig. 4. Data compression procedure

C. *Threaded emulator*

In our implementation, some operations are grouped to be executed incessantly. The program is divided into some blocks [6]. The word "block" refers to each section of code that has one entry point and one exit point. All instructions inside a block do not require to be fetched one by one at runtime. It is possible to run the whole block with just one fetch and then decode and execute the operations inside the block.

There are two schemes to implement it: 1) switch case scheme that uses C's switch case implementation and 2) function pointer accesses pre-stored entry points in a structure. Both of these methods were not actually beneficial in ARM processor and we did not gain any performance using them. The implementation of each scheme and reasons of their failure are explained below in each section.

1) *Switch case*

Here, entry instruction of each block is accessible through a big switch case. In each "case" the whole block is executed unceasingly until it reaches the exit point and program counter is increased for all instructions. Figure 5 shows the implementation of such a scheme. An explanation of the performance degradation can be register pressure or the increase of cache miss rate.

2) *Function pointer*

Another scheme is to keep the record of entry points of each block as a function pointer and a set of these function pointers is kept for later references. Figure 6 shows an implementation of this method. This method was a bit faster than *switch case* method, however, for the same reasons, it causes overall performance deterioration.

```
switch(state.pc)
{
case0x1a32:{
     uint16_t offset1=(state.d<<8)|state.e;
     state.a=state.memory[offset1];

     uint16_t offset=(state.d<<8)|state.e;
     state.a=state.memory[offset];
     WriteToHL(&state,state.a);

     state.e++;

     if(state.e==0)
          state.d++;
     state.l++;
     if(state.l==0)
          state.h++;
     state.b-=1;
     FlagsZSP(&state,state.b);
     state.pc+=5;
     uint8_t *opcode=&state.memory[state.pc];
     if(0==state.cc.z)
          state.pc=(opcode[2]<<8)|opcode[1];
     else
          state.pc+=2;

     cycle+=39;
     cycle+= Emulate8080Op(&state);
     break;
}
```

Fig. 5. Onde decoded block in switch case emplementation

```
int pc15c7()
{
     /*pc number 15c7 entry always perform
     three instructions as follows*/

     //read from HL
     state.a=ReadFromHL(&state);

     //AND A with A
     state.a=state.a&state.a;
     LogicFlagsA(&state);
     state.pc+=2;

     //Jump Zero
     opcode=&state.memory[state.pc];
     state.pc++;
     if(0==state.cc.z)
          state.pc=(opcode[2]<<8)|opcode[1];
     else
          state.pc+=2;

     //returns cycles of this block
     return21;

}
```

Fig. 6. One decoded block in function pointer implementation

D. System level optimization

There are 12 general purpose registers available in ARM-11 processor that can be used by the programmer. When we reviewed the assembly code generated by the compiler, we discovered that the compiler had used only few registers, while more registers are available in the destination hardware. Our first scheme was to use inline assembly in C code. Most of the program was still coded in high-level language but some functions that have more execution rate were implemented in assembly language. The disadvantage of this method is the additional code inserted by compiler to handle communication between the C code and inline assembly parts. The additional code increases the overall runtime of emulation.

Our second approach was to implement our emulator in assembly from scratch. The advantage of this implementation is a result of managing hardware registers manually. One good idea is to have most accessed values in register file instead of main memory. Another advantage of this scheme is that the number of instructions is reduced. However, due to time constraints in the contest we did not have enough time to implement this method.

IV. EXPRIMENTAL RESEULT

Results of three different general approaches tested with four Replay inputs are listed in Table 1. *Original* label refers to the reference program provided by contest designers. *Modified* refers to the combination of techniques applied by our approach to get the best result and *Modified** refers to the results of *modified* approach using overclocking of Raspberry Pi. Figure 7 shows the same results in a graphical way for the sake of comparison.

TABLE I. PERFORMANC RESULTS

Execution time of Replays on three approaches	**Game replays**			
	Game-960	**Short**	**Kyle**	**Long**
Original	4.63	0.51	12.50	27.54
Modified	2.95	0.32	7.89	17.79
Modified*	2.03	0.23	5.40	12.16

V. CONCLUSION

In this paper, we described some techniques to decrease the running time of Intel's 8080 emulator on a Raspberry Pi device. The aim of the emulation was running an old arcade game, Spade Invaders, on an ARM-based emulator. We took advantage of function call improvements and modified data structure of the reference implementation to achieve a reasonable performance. This problem was designed for the software contest of MEMOCODE 2014 and the problem constraints hindered further optimizations. Experimental results showed that our approach could obtain a speedup of 2.5x over the reference implementation that could rank us as second in the contest.

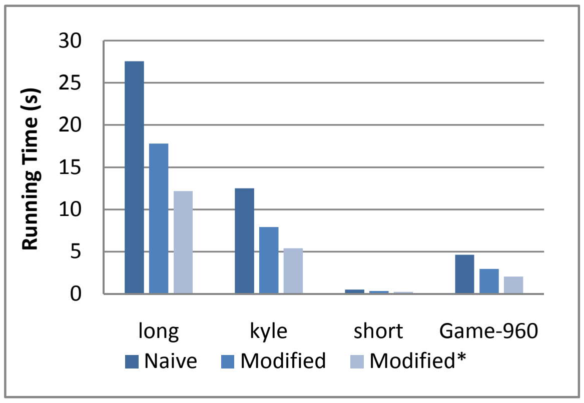

Fig. 7. Running time of four different replays with three approaches

ACKNOWLEDGMENT

We are grateful to Prof. Sarbazi-Azad, head of the School of Computer Science of the Institute of Research in Fundamental Sciences (IPM), for his supports and useful guidance. We are also grateful to the researchers of the school for their helpful discussions and suggestions.

REFERENCES

[1] V. M. d. Barrio, "Study of the techniques for emulation programming," FIB UPC.

[2] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," in *ACM SIGMETRICS Performance Evaluation Review*, 1996, pp. 68-79.

[3] (2014). *MEMOCODE Software Design Contest 2014*. Available: https://caesr.uwaterloo.ca/memocode/

[4] *Emulator101*. Available: http://emulator101.com/

[5] *Raspberry Pi device*. Available: http://en.wikipedia.org/wiki/Raspberry_Pi

[6] *Threaded code*. Available: http://www.complang.tuwien.ac.at/forth/threaded-code.html