Program Analysis and Transformation

Final Assignment

Benjamin Brandt Ohrt, zpn492

May 31, 2019

Abstract

The Pendulum Instruction Set Architecture (PISA) was first introduced in 1995 by Carlin James Vieri and is a reversible assembly language. The assembly language has later been improved and several high-level languages has been build upon it. One of those languages is the extension to the reversible object-oriented programming language (ROOPL) ROOPL++ presented in 2018. One of the issues with ROOPL++ was the amount of produced target code. In this paper we compile source code for the data structure of a binary tree, analyse the possibility of redundant target code and points out where an optimization could be done.

Contents

1 Introduction

1.1 Assembly language

A CPU understands its own machine language. Instructions in machine language are numbers stored as bytes in memory. Each instruction has its own unique numeric code called its operation code or opcode for short. An example is the instruction that says add EAX and EBX registers together and store the result back into EAX is encoded by the following hex code:

03 C3

This not readable for the blunt eye. Which is why we have a program called an assembler. An assembler is a program that reads the text of a assembly language program and converts the assembly into machine code. [6, p.11].

The assembly language origins back to Birkbeck College 1946-1962, where the creation was credited to Kathleen Booth.[1] Kathleen Booth was a PhD whom both visited with John Von Neumann with the famous The Von Neumann Architecture[3], helped with the design of multiple machines and was one the founders of the Birkbeck department of computer science.

The assembly language is a symbolic programming language, closest to machine code.[2] An assembly language program is stored as plain text. The text consists of a set of instructions which is processed chronological. Each assembly instruction is equal to one machine instruction for example could above addition be shown as:

add EAX EBX

With the assembly language words as add can be used as mnemonic for the instruction add and so fourth. [6, p.11].

1.1.1 Example program

```
j main                          # for int i = 0; i < 100; i++
                                #    count += i * 2
loop:
mul $t0, $a1, 2
add $a0, $a0, $t0               # count += i * 2
addi $a1, $a1, 1                # i++
slt $t0, $a1, $a2
beq $t0, 1, loop                # i < 100
jr $ra                          # return

main:
addi $a0, $zero, 0              # count = 0
addi $a1, $zero, 0              # i = 0
addi $a2, $zero, 100            # max
jal loop
```

Small example program written in MIPS assembly

1.2 Reversible computing

A reversible computing system has, at any time, at most a single previous computation state as well as a single next computation state, and thus a reversible computing system.[7] It comes with the promise of reduced energy dissipation, when erasure of information is left out of the program.

The inspiration of reversible computing dates back to 1867 and the study of thermodynamics where James Clark Maxwell made a thought experiment also known as Maxwell's Demon. In the experiment he questions that of the second law of thermodynamics which says:

> [..] It is impossible in a system enclosed in an envelope which permits neither change of volume nor passage of heat, and in which both the temperature and the pressure are everywhere the same, to produce any inequality of temperature or pressure without expenditure of work.[8, p.16]

The experiment was later published in 1872 in a book by Maxwell: Theory of Heat[11], but the connection between thermodynamics and computations was first seen in 1949 by John von Neumann, where he talks about a computation must have a minimum thermodynamic energy dissipation which he determines to[5, p.20][8, p.18]:

> k_BTlnN, k_B as the Boltzmann's constant, T as the temperature and $N = 2$.

Rolf Landauer realize that modern computers with irreversible processes must dissipate that minimum energy as von Neumann described if the erasure of a bit occures thus resulting in atleast the increase of entropy by k_Bln2.[5, p.20][8, p. 18] He also states that reversible operations does not produce the dissipation and irreversible operations can avoid the dissipation by storing information of the computational history. At last Landauer states that the stored history must be erased irreversibly which would just postpone the dissipation of entropy, That last statement is later proven wrong by Charles Bennett.[8, p.19]

In 1970 Charles Bennett looks at Landauers last statement and decides to make an experiment. He creates a reversible program consisting of two halves. The first which provided the calculations intended and the second which undid the calculations from the first halves thus ending the program in its starting point. In the experiment he uses a Turing Machine[10] and shows that the information produces by the intended calculation could be used to undo the calculation instead of just be thrown away. With that experiment Bennett had shown that a reversible process would not just postpone the dissipation of energy it would be able to remove it from the equation.[8, p.18]

From Bennetts experiment he came up with a description for an "enzymatic Turing machine" in which reversible logically operations could be executed[8, p.19]. To fullfill the promise of reduced energy dissipation their was final need of physically logic devices to perform the reversible operations on. Fredkin, Toffoli and Feynmann all aided this need by introducing reversibly logic-gates.[8, p. 19, 22]

1.3 Pendulum microprocessor

Pendulum is a reversible microprocessor, invented by Carlin James Vieri at MIT in 1995. The invention took offset in the existing MIPS R2000 architecture.[REF til ARK][4, p. 29] Vieri's motivation was to create a reversibly processor which would avoid destruction of information and reduce the energy dissipation shown in thermodynamics[1.2] and discussed by Charles Bennet.[8] For this he would focus on memory access, datapath operations on stored values and control flow operations.

The processor has three registers for used in the control flow:

1 The program counter (PC) for storing adress of the current instruction

2 The branch register (BR) for storing jump offsets

3 The direction bit (DIR) for keeping track of the execution direction.

PC is incremented/decremented by the value stored in DIR, thus changing DIR between 1 & -1, will decide in which direction the instructions will be executed. When the DIR is -1, all instructions is inverted.[4, Vieri - Chapter 3, p. 21]

Control flow - Branch instructions like branch or jump is in normal architectures not reversible, that is because the come-from instruction is not stored anywhere, thus the goto instruction not knowing who called it. This could be handled by storing the PC just before the branch/jump in a special register. In later versions of the Pendulum ISA architecture paired branches is introduced, such that each goto instruction should have branch instruction to the come-from.

Control flow - If/then statements is in need of an exit condition known as assert.

if e1 **then** s1 **else** s2 **fi** e2

Control flow - for-loop is in need of an entry, loop and exit condition. Where the e1 should only be true upon entry and e2 only on exit.

from e1 **do** s1 **loop** s2 **until** e2

Control flow - subrutines in encased between a top and a bottom, which both contains a branch to eachother. It follows that the subrutine is skipped when executing instructions sequentially.

top: BRA bot

<..rutine definition goes here..>

bot: BRA top

Memory access is always an exchange. The exchange instruction swaps register value with the value in memory at an address specified by another register.[4, p. 32]

1.4 Motivation

1.5 Outline

2 Programming languages

2.1 Pendulum Assembly Language

Pendulum Instruction Set Architecture(PISA) Assembly Language or a shorthand PAL, is a reversible language. It assembles by the Pendulum microprocessor.[4, p. 48] In this paper we use the Pendulum virtual machine PendVM[9], which executes programs written in PAL. Pendulum instructions is almost identically to the conventional RISC(Reduced Instruction Set Computing) processor. Thus reversible it introduces some new features.

Branch register is normally zero, if the branch register is not zero, the PC increments by the value in the register.[REF - PENDULUM 248] When is happens it the instruction at the destination executes and the branch register is cleared. When implementing subroutines one must use SWAPBR which allows direct access to the branch register, and therefor an exchange the value. It must follow that the subrutine negates the value in the branch register, such that the next SWAPBR will branch back to the location it came from. The branch at the location cancels out the branch register and the PC continues sequentially. SWAPBR has also the capability of performing switch statements. [REF - PENDULUM p. 281] START and FINISH marks the beginning and end of a program. BRA is an unconditional jump.

Small PAL program:

```
;; pendulum pal file

;; Label      Instr  Args          ; Pseudocode description
;; ---------  -----  ----          ; -------------------------
start:     START
           ADDI $4 43
           ADDI $4 -43
           FINISH                  ; EXIT PENDVM1
```

Pendulum Assembly program

2.2 Reversible Object-Oriented Programming

Reversible Object-Oriented Programming (ROOPL) and its predecessor ROOPL++ which expanded the language with dynamic memory was created by Tue Haulund and Martin Cservenka. It is build with inspiration from the reversible programming language Janus and compiles to the Pendulum Assembly language.

2.2.1 Malloc

malloc

```
1  l_malloc_top:          BRA     l_malloc_bot
2  l_malloc:              SWAPBR  $2
3                         NEG     $2
4                         ADDI    $9  2
5                         XOR     $8  $0
6                         ADDI    $1  1
7                         EXCH    $6  $1
8                         ADDI    $1  1
9                         EXCH    $7  $1
10                        EXCH    $2  $1
11                        ADDI    $1  -1
12                        BRA     l_malloc1
13                        ADDI    $1  1
14                        EXCH    $2  $1
15                        EXCH    $7  $1
16                        ADDI    $1  -1
17                        EXCH    $6  $1
18                        ADDI    $1  -1
19                        XOR     $8  $0
20                        ADDI    $9  -2
21 l_malloc_bot:          BRA     l_malloc_top
```

malloc1

```
1  l_malloc1_top:         BRA     l_malloc1_bot
2                         ADDI    $1  1
3                         EXCH    $2  $1
4                         SUB     $17 $8
5                         XOR     $17 $4
6  l_malloc1:             SWAPBR  $2
7                         NEG     $2
8                         ADD     $15 $7
9                         XOR     $15 $9
10 l_malloc1_bot:         BRA     l_malloc1_top
```

3 Analysis of redundancy

In this is section we analyse the possibility of redundant target code after compilation of a ROOPL++ program. First we define a smaller area of *redundancy*:

> Redundancy is several immediate instructions in a row which all affect the same register. For example is the two instructions below seen as redundant:
>
> ADDI $3 10
>
> ADDI $3 8
>
> Instead the target code could have used a single instruction:
>
> ADDI $3 18

Now lets look at a successfull analysis:

> Identify redundancy (if any) in the target code
>
> Describe a method for reducing redundancy.
>
> Show that a program p written in ROOPL++ have the same result as the program p' which is copy of p but with above method applied.

3.1 Reduction

For the task of a successfull analysis we need an algorithm in which can find *redundancy*, merge and remove instruction and atlast update the stack point with the new immediate value. This algorithm can be applied to an existing PAL program, which means, that is first useful, when a ROOPL++ program has been compiled to target code. Thus the algorithm must take a .PAL file and produce a .PAL file.

```
remove_redundancy(filepath fp):
    /* read is a function which takes a filepath
     * and returns a array of elements, where each
     * element is a line the the file.
     * map(func1, l[]) applies func1 to every element of l */

     lines = map(parse_inst, read(fp))

     lastRegister = ''
     lastInstruction = 0
     removeableLines[]

     for i = 0 to length(lines)
       if lines[i][0] == 'ADDI' then
         if lastRegister == lines[i][1] then
            lines[lastInstruction][2] += lines[i][2]
            insert(removeableLines, i)
         else
            lastRegister = lines[i][1]
            lastInstruction = i
       else
           lastRegister = ''
           lastInstruction = 0

     /* Remove all redundant instructions, update stack pointer */
     lines = removeLinesAdjustSP(lines, removeableLines)

     /* write is a function which takes an array
      * and writes a line for each element to a named file */

     write(lines, 'reduced.pal')

/* A method to divide an instruction
 * into opcode, arg0, arg1, arg2 */
parse_inst(instruction):
   return split(instruction, ' ')
```

3.2 Tools

PendVM[9] is a virtual machine on which we can run PAL programs. It does not support methods to save the results of an executed program, which is why i have expanded one of the opcodes: OUT.

OUT is now able to write or append a register value to the text-file static_ storage.text. For now ROOPL has not been expanded which means, that using the OUT functionality requeries three steps:

The immediate value in ADDI just after START has to incremented by the value of need OUT instructions.

The immediate value in ADDI just before FINISH has to incremented by the value of need OUT instructions.

OUT instruction has be manually inserted into the resulting .pal file of a compiled ROOPL++ program.

The expansion of PendVM exists of the following files:

```
int i_out(WORD, WORD, WORD);
```

pendvm.h

```
("OUT", {REG, REG, NIL}, i_out)
```

pal_ parse.c

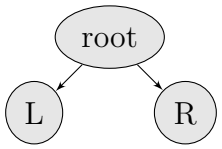
```
int
i_out(WORD r, WORD u1, WORD u2)
  {
  char buf[..];
  sprintf(buf, '%d', m->reg[r1]);
  const char *p = buf;

  FILE *file;
  file = fopen('static_storage.txt', m->reg[r2]);
  fputs(p, file);
  fclose(file);
  return 0;
  }
```

machine.c

4 Data structures

4.1 Binary tree

A binary tree in computer science is a data structure, which is either empty or consist of a root node, where each node has atmost two children. You could see the binary tree as a tuple of the single node root and the two binary trees L & R.



Simple binary tree, with the a root node and two binary trees L & R as children to the root.

Their are different types of binary tree all with their own properties. fx A full binary tree is one where every node either have zero or two child-nodes. A perfect binary tree is one, where all interior nodes have two children and all leaves have none. the A balanced binary tree is one where L & R of every node only differ in height with no more than one. [REF-Introduction to algorithms]

Properties of binary tree is fx maximum height, number of nodes, number of leafs and so on. Operations of different sort can be used of a binary tree some of em is:

Traversal: Search for a node

Insertion: Insert a new node

Deletion: Remove an existing node

5 Conclusion

6 Further work

6.0.1 Instruction operands - [OVERVEJ]

Machine code instructions vary, but in general each instruction itself will have fixed number of operands (0 to 3). Operands can have the following types:[6, p. 12]

register: These operands refer directly to the content of the CPU-register.

memory: These refer to data in memory. The address of the data may be a constant hardcoded into the instruction or may be computed using values of registers. Address are always offsets from the beginning of a segment.

immediate: These fixed values that are listed in the instruction itself. They are stored in the instruction itself (in code segment), not in the data segment.

implied: These operands are not explicitly shown. For example the increment instruction adds one to the register or memory. The one is implied.

6.0.2 Pendulum operation

Operation type					
R	op	rsd	rs	sh/rot	func
bit	6	5	5	5	11
J	j/cf	Target			
bit	6	26			
B	j/b op	ra	rb	offset	
bit	6	5	5	16	
I	op	rsd	immediate		
bit	6	5	21		

Jump instructions other than j and all conditional branches uses B-type instruction encoding and specify two registers and an offset.

Immediate instructions, I-type, specify one register and a 21 bit signed or unsigned, depending on the opcode, immediate value.

Instruction	
ADD	
SUB	
ADDI	
AND	
ANDI	
BEQ	
BGEZ	
BGEZAL	
BGTZ	
BLEZ	Branch on less than or equal to zero
BLTZ	Branch on less than zero
BLTZAL	Branch on less than than zero and link
BNE	Branch on not equal
CF	Come-from

6.1 Pendulum

, with exception of two constraints:

> Memory access is always an exchange. The exchange instruction swaps register value with the value in memory at an address specified by another register.[4, p. 32]

> Non-sequential instructions like jumps to another macro must have a return address. This is needed to keep the reversibility. [4, p. 36] Say an instruction is reached by one jump, then that instruction only needs to keep track a single destination. In case an instruction is reached from multiple other instructions, it needs to keep track of which instruction that made the jump.[4, p. 25]

References

[1] http://www.computinghistory.org.uk/det/32489/Kathleen-Booth/

[2] https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.asma400/asmr102112.htm

[3] https://en.wikipedia.org/wiki/Von_Neumann_architecture

[4] Vieri, C. J. et al. *Pendulum: A Reversible Computer Architecture*. Master's Thesis. University of California at Berkeley 1993.

[5] Michael P. Frank. *Reversibility for Efficient Computing*. Ph.D Thesis. University of Florida, 1999.

[6] Carter, P. A. *PC Assembly Language*. 2006

[7] Yokoyama, T. and Glück, R. *A reversible programming language and its invertible self-interpreter*. ACM, 2007.

[8] C. H. Bennet. *Notes on the history of reversible computation*. IBM J. Res. Dev., 32(1), 1988

[9] https://github.com/TueHaulund/PendVM

[10] https://plato.stanford.edu/entries/turing-machine/

[11] J. C. Maxwell. *Theory of Heat,* 4th Ed., Longmans, Green & Co., London, 1875 (1st Ed. 1871)