Revision 1.0.1

1964 Recompiling Engine Documentation

Designed as Companion Source Code Documentation for 1964 Version 0.6.0

Chapter
1

Introduction

"The Skinny"

What is this document?

This document gives a detailed technical explanation of how the 1964 dynamic recompiling engine works.

- Chapter 2 explains the basic concepts involved in creating a recompiler.
- Chapter 3 details how to compile opcodes.
- Chapter 4 explains the 1964 source code implementation.

To better understand how to do something, it is important to first know why you are doing it. In explaining the dynamic recompiler, I have taken this approach to explain "Why is it done?" Then, "How is it done?"

For whom is this document?

This doc is for anyone who is interested in understanding how the dynamic recompiler in 1964 works. It is assumed that you already have some understanding of MIPS assembly and Intel 80x86 assembly. This doc can also be useful for some other emulation projects that don't use a recompiler. If you apply the concepts presented in this doc in another app, just give a little mention to me (schibo) and I'll be happy.

Chapter
2

Basic Concepts

“Insert Gerbil A Into Slot B”

Why write a recompiler?

In the wonderful world of emulation, you may have heard the terms “interpreter” and “dynarec” thrown around. To put it simply, an interpreter will be slower than a “dynarec” or dynamic recompiler. The only advantage of a recompiler is speed. A recompiling core will not give you improved compatibility, better graphics, etc. Just more speed.

An interpreter performs the following tasks:

1) Fetch a 32bit MIPS instruction.

2) Parse the instruction so that you know what the opcode is. If it is an arithmetic opcode such as ADD for instance, you must also do additional parsing of the instruction to determine what registers are being used.

```
unsigned char __opcode = (unsigned char)(Instruction >> 26); // opcode
unsigned char __rd = ((Instruction >> 11) & 0x1F);           // rd register
unsigned char __rt = ((Instruction >> 16) & 0x1F));          // rt register
unsigned char __rs = ((Instruction >> 21) & 0x1F));          // rs register
```

Fig. 2-1 Parsing

* Note: An ADD instruction is known as a special instruction, so __opcode will be zero. This means we need to take an additional step to parse the last 5 bits of the instruction to get the opcode.

3) Execute the opcode. This involves loading the operand registers, executing the opcode such as ADD, then writing the result to the destination register.

```
void r4300i_add(_int64* rd, _int32* rs, __int32* rt) {
rd = (_int64)(rs + rt);
}
```

Fig. 2-2 Sample interpretive opcode function

* Note: The MIPS r4300i processor has 32 64bit General Purpose Registers. Since in this case the ADD opcode performs a 32bit operation with sign extend, we need only retrieve the low 32bits of the rs and rt registers, then sign extend the result for the rd register.

4) Check for interrupts.

5) Repeat.

```
If you like, take a look at the following code in 1964. You will see that the
interpreter performs the 5 steps.

- RunTheInterpreter() in emulators.c

- Fpu.c … lotsa opcodes

- R4300i.c… lots more opcodes
```

Whew, that's a lot of steps and is gonna be slowwww! But that's basically all that's involved in writing an interpretive core. However, the moral of this story is that "dynarec", as we affectionately call it, **is just 1 step most of the time: execute code**! This is, of course, after it has been compiled to native 80x86 code. ☺

How do I start ?

Remember, what we're looking to do is get speeed. So the first thing we're going to need to do is try to get rid of the overhead of these interpretive steps.

"But schibo, I've written 80,000,000 opcode functions in my interpreter. Is all that work going to waste now that I'm writing a recompiler?"

"Hell no! Dat be our backbone!"

Here's how we get our feet wet. Take a look at this assembly code:

```
;add
push rt
push rs
push rd
call r4300i_add
;sub
push rt
push rs
push rd
call r4300i_sub
. . .
```

Fig. 2-3 Your first dyna skeleton.

What that represents is a bit of newly recompiled code that our dynamic recompiler generates. When we call this code, the code will call the interpretive functions. You can see how we've eliminated tasks 1 and 2 of the interpreter (parsing is now done at compile-time only), and we already know what opcode function needs to be called by the time the code is ready to be executed. Task 5 in essence is gone also, since code by its nature is performed sequentially ☺.

Task 4 however, is a bit trickier. In 1964, instead of checking for interrupts after each instruction (as in the interpreter), interrupts are checked at the end of each block[1] of code in the dynarec. While this may not be the most accurate approach, it is certainly faster than checking for interrupts after each instruction.

The code immediately above uses the __cdecl calling convention. If you are unfamiliar with calling conventions, you can disregard the next 2 paragraphs, as long as you are building in C and your compiler is using __cdecl, which should be the default convention.

For some more speed in C, you may use __fastcall to call your opcode functions. With _fastcall, you will move rd to ecx, rs to edx, then push the remaining parameters in filo order. In the case of our ADD example, just push rt.

If you have your opcode functions as members of a C++ class, you will need to use the __thiscall[2] calling convention. __thiscall uses an "invisible" *this* parameter. For this case, you will need to first move the value of the address of the class to the ecx register. Then you can push your parameters rt, rs, rd.

How was that code generated?

Have a look at RunTheRegCache() in the file emulators.c in 1964 0.6.0 source.

This function is the heart of the dynarec.

What it does:

- Recompile block of code if the block has not yet been compiled.
- Call the block.
- Check for interrupts and handle other tasks.

```
void RunRecompiler() {
char* BlockStart;
     while (1) {
          BlockStart = LookupTable[pc]; // The lookup table will give us the
block's start address
          if (BlockStart == NULL) {
               BlockStart = RecompileTheBlock();
               LookupTable[pc] = BlockStart;
```

[1] In this case, a "block" refers to code that is terminated by a MIPS jump, branch, or eret instruction.
[2] Calling conventions in Microsoft Visual C++. Refer to your C/C++ compiler documentation for additional information.

```
            __asm call BlockStart;
            CheckInterrupts();
        }
    }
```

Obviously, a 1-D lookup table like the one shown in the above pseudocode would take too much memory. Your best bet is to make a 2-D lookup table:
LookupTable[<*segment*>][<*offset*>]. Then you can do something like this:

```
BlockStart = LookupTable[pc>>16][(unsigned short)pc];  // The lookup table
will give us the block start address
```

In our pseudocode, RecompileTheBlock() is the magic function that does the compiling.

Notice that the first 2 steps of the function are the same as the interpreter:

1) Fetch a 32bit MIPS instruction.

2) Parse the instruction so that you know what the opcode is. If it is an arithmetic opcode such as ADD for instance, you must also do additional parsing of the instruction to determine what registers are being used.

3) Compile the instruction.

4) Repeat until end of block. A block is terminated by a MIPS jump, branch, or eret instruction.

5) Return the start address of the block of new code.

Now we can generate code similar to that in **Figure 2-3**. If you look at dynacpu.c in the 1964 source code, you will see a copy of all the opcode functions that were in r4300i.c (r4300i.c is the interpreter). Whenever you see a dyna opcode function using the macro INTERPRET(opcode), it is creating code to call it's respective interpretive function in r4300i.c. When the block is compiled and ready for execution, RunRecompiler() calls BlockStart.

Chapter
3

Compiling Opcodes

“Using the Native Tongue”

Inline your opcode function calls

Now that we have a dynarec.c file with opcode functions that call interpretive ops like this,

```
void dyna_r4300i_add() {
     INTERPRET(r4300i_add);
}
```

wouldn’t it be nice to eliminate the overhead of the function calls that we see in **Figure 2-3** and just execute recompiled code? That’s our goal.

Have a look at this pseudocode:

```
_int64 GPR[32]; // The 32 MIPS General Purpose Registers
void dyna_r4300i_add() {
     MOV_MemoryToReg(1, Reg_EAX, &GPR[__rs]);  // load
     MOV_MemoryToReg(1, Reg_ECX, &GPR[__rt]);  // load
     ADD_Reg2ToReg1(1, Reg_EAX, Reg_ECX);
     CDQ();
     MOV_RegToMemory(1, Reg_EAX, ModRM_disp32, &GPR[__rd]);   // store
     MOV_RegToMemory(1, Reg_EDX, ModRM_disp32, &GPR[__rd]+4); // store
}
```

Fig. 3-1 Your first recompiled opcode function

So instead of calling r4300i_add, we’ve essentially made r4300i_add inline, and our recompiled code from Figure 2-3 now looks like this:

```
; add
mov eax, dword ptr GPR[rs]
mov ecx, dword ptr GPR[rt]
add eax, ecx
cdq ; convert double (32bit) to quad (64bit). eax is sign-extended and the
high 32bits is stored in edx
```

```
mov dword ptr GPR[rd]+4, edx
;sub
push rt
push rs
push rd
call r4300i_sub
. . .
```

Fig. 3-1-1 The recompiled code

The new functions you see in the body of dyna_r4300i_add() in **Figure 3-1** were written by Lionel of N64VM. They are great functions that read like assembly and help you to write recompiled code. In 1964, they are in X86.c and X86.h.

Cache your registers

Thus far, we've eliminated a lot of overhead, but we can still optimize so much more. You will notice in **Figure 3-1** that the ADD code we built performs the following:

1) Load 32 bits
2) Load 32 bits
3) ADD
4) CDQ
5) Store 32 bits.
6) Store 32 bits.

The idea behind **register caching** is to reduce the steps so that we only do step 3. Beautiful recompiled code would look something like this,

```
add eax, ecx ;add
sub ecx, edx ;sub
...
```

Of course, we still need to load and store the emulated MIPS registers. The key here though is *when*. We want to do that as infrequently as possible! Loading from memory and storing to memory are extremely expensive operations.

Look at register caching this way: if we have a sequence of MIPS opcodes that use the same MIPS registers, a lot of the memory loading and storing that we do when we emulate the opcodes are a waste, because in many cases we already have the register data in our Intel processor, so there's no need to store this data to memory only to load in the same data again.

To eliminate this overhead takes some crafty thought. For starters, we will use a single-pass regcaching algorithm.

Your first register-caching algorithm

Here is the pseudocode to explain the logic of your compiler’s algorithm as it runs. (For simplicity, we will assume the registers are 32bits and __rd, __rs and __rt of a MIPS instruction are each unique.)

```
//Start of block
...
/////////// ADD opcode
if rs is not assigned (mapped) to an Intel x86 register
{
     // Assign rs to an available Intel x86 register
     x86reg[0].mips_reg = __rs; // x86reg[] is our regcache map
     Compile the assignment: -> mov eax, dword ptr GPR[__rs]
}
if rt is not assigned to an intel register
{
     // Assign rt to an available Intel x86 register
     x86reg[1].mips_reg = __rt; // the array index value is 1 (ecx) just to
illustrate flow
     Compile the assignment: -> mov ecx, dword ptr GPR[__rt]
}
if rd is not assigned to an Intel x86 register
{
     // Assign rd to an available Intel x86 register
     x86reg[2].mips_reg = __rd;
}
Compile the ADD -> add eax, ecx

/////////// SUB opcode
if rs is not assigned to an Intel x86 register
{
     // Assign rs to an available Intel x86 register
     x86reg[3].mips_reg = __rs;
     Compile the assignment: -> mov edx, dword ptr GPR[__rs]
}
if rt is not assigned to an intel register
{
     // Assign rt to an available Intel x86 register
     x86reg[4].mips_reg = __rt;
     Compile the assignment: -> mov ebx, dword ptr GPR[__rt]
```

```
if rd is not assigned to an Intel x86 register
{
     // Assign _rd to an available Intel x86 register
     x86reg[5].mips_reg = __rd
}
Compile the SUB -> sub ecx, edx
...

//End of a block. Time to Flush All Registers.
// Unmap all the mapped registers and store (flush) them back to memory.
for (k=0; k<8; k++)
{
     if (x86reg[k].IsDirty)
          // Compile the write-back to memory:
               -> mov dword ptr GPR[x86reg.mips_reg], eax<-(this argument
depends on k)
         x86reg[k].IsDirty = 0;
}
ret ; returns to the recompiler
```

As you start to develop this new regcache code, you'll need to flush the registers just before the executed compiled code jumps out of itself.

We only flush the dirty registers. A register is marked as "dirty" if its data is to be modified. In our ADD opcode, for instance, we have rd = rs + rt. In this case, rd is the dirty register because it is the only one that is modified. A register is not unmarked as "Dirty" until it is "cleaned". This is when it is flushed back to memory.

Why do I need to flush?

The reason we need to flush is because when it resumes, recompiled code is not going to know or care what your compiler's register map looks like. We don't want the x86 registers to refer to the wrong MIPS registers.

Generally, we will flush in 3 cases:

1) At the end of a block when an x86 RET instruction is encountered.

2) Whenever you use the x86 CALL instruction to call a separate section of code. Examples of this occurrence include:.

    a. When an interpretive opcode is called.

   b. When an opcode exception of a reg-cached opcode needs to be processed.[3]

3) When we want to map another register, but all the x86 registers are already assigned to a MIPS register. In this case, we need to only flush one register to free up a register.

If you are calling a function from your dynarec and you know that the registers that are cached will not be modified by the code you are calling, you can do following instead of flushing all the registers:

```
pushad
call TheFunction
popad
```

Fig 3-2 Calling an interpretive op when some of our code is regcached

PUSHAD pushes the contents of the 8 x86 registers onto the stack.

When we return from the call, we simply pop those values with POPAD. No flushing ☺. We can continue to use our regcache map.

How do I cache 64bit MIPS registers?

This adds an additional level of complexity. The MIPS processor we are emulating is 64bit. As you know, our Intel processor has 32 bit registers. Fortunately, for MIPS opcodes that do 32bit logic such as add, sub, subu, addi, etc, we can perform these opcodes as 32bit. Then when it's ultimately time to flush, we simply sign extend the 32bit dirty registers.

```
; flush: ecx represents the result of 32bit logic and is dirty.
mov dword ptr GPR[x86reg[1].mips_reg], ecx ; store the lo 32bits to memory
sar ecx, 31 ; sign-extend the register
mov dword ptr GPR[x86reg[1].mips_reg]+4, ecx ; store the hi 32bits to memory
ret
```

Fig 3-3 Flushing 32-bit logic

* Notice that instead of using CDQ, we use SAR ECX, 31. The advantage of using SAR instead of CDQ is that we can calculate the high 32bits of ecx without overwriting the eax and edx registers. There might be unflushed data in eax and edx, and we don't want to overwrite it. Besides, CDQ is a slow unpairable instruction.

However, there are MIPS opcodes that perform true 64bit logic. For these opcodes, it is unavoidable to reserve 2 x86 registers for each MIPS register that is used in the instruction. Examples include: OR, DADD, and DSUB. Flushing would look like this.

[3] For information about exception handling, refer to zilmar's emubook.

```
; flush: ecx represents the low dword result of 64bit logic and is dirty.
mov dword ptr GPR[x86reg[1].mips_reg], ecx ; store the lo 32bits to memory
; flush: ebx represents the corresponding high dword result of 64bit logic.
mov dword ptr GPR[x86reg[1].mips_reg]+4, ebx ; store the hi 32bits to memory
ret
```

Fig 3-4 Flushing 64-bit logic

For the 64bit logic, our register map array now will need to keep track of which two x86 registers the low and high dwords of a MIPS register are mapped to.

Chapter
4

Examining the 1964 Source Code

“Using the 1964 Tongue”

My chicken scratch

We left off in chapter 3 discussing briefly what our register map needs to keep track of.

```
typedef struct x86regtyp
{
    _u32 BirthDate;
    _s8  HiWordLoc;
    _s8  Is32bit;
    _s8  IsDirty;
    _s8  mips_reg;
    _s8  NoNeedToLoadTheLo;
    _s8  NoNeedToLoadTheHi;
    _s8  x86reg;

} x86regtyp;
x86regtyp x86reg[8];
```

Fig. 4-1 Regcache structure in regcache.h

Figure 4-1 shows our register map structure. The x86reg[] array has 8 x86regtyp items..one for each x86 register. The array index indicates which x86 register is used.

- x86reg[0] refers to the eax register.
- x86reg[1] refers to the ecx register.
- x86reg[2] refers to the edx register.

And so on.

The fields of each array item indicate attributes of an x86 register. Here are examples of the syntax with x86reg[2]:

x86reg[2].mips_reg;

Indicates which MIPS General Purpose Register (0-31) is mapped to EDX. This corresponds to the low 32bits of the MIPS register.

If EDX is unused, x86reg[2].mips_reg = -1;

x86reg[2].HiWordLoc;

Indicates which x86 register holds the high dword of the MIPS register.

If the HiWordLoc field is the same as the array index (in this case 2), then EDX represents a MIPS register that used 32bit logic. When the register is flushed (if the register is marked as dirty), we will store it like you see in **Figure 3-3**.

If the HiWordLoc field is *not* the same as the array index, then EDX represents a MIPS register that used 64bit logic, and the HiWordLoc field specifies the x86 register that corresponds to the high dword of the MIPS register. When the register is flushed (if the EDX register is marked as dirty), we will store it like you see in **Figure 3-4**.

x86reg[2].IsDirty;

Indicates whether or not the mapped register will need to be stored to memory when it is flushed.

x86reg[2].NoNeedToLoadTheLo, x86reg[2].NoNeedToLoadTheHi;

Indicates whether or not we need to load the contents of the register from memory when we map it. Generally, we set these fields to 1 when we know that the register being mapped (like rd in ADD) is the dirty result of an operation, and it is not an operand (like rs, rt in ADD). We need to take care when using these fields to make sure that __rd is not equivalent to either __rs or __rt. Otherwise, __rd represents both an operand and a result, in which case we need to load the data from memory.

xRT->x86reg

Indicates which x86 register is used. Obviously, this is ambiguous in our register map, since we already know that the array index refers to the x86 reg. However, I use the x86regtyp struct in my dyna opcodes like this also:

x86regtyp xRT[1]; // gets and sets rt attributes

x86regtyp xRS[1]; // gets and sets rs attributes x86regtyp xRD[1]; // gets and sets rd attributes.

etc..

So, the MapRegister() function in regcache.c does our mapping. From the dynarec opcode functions, I pass xRT[], etc..as a parameter into MapRegister().

Before calling MapRegister, I set the xRT[]'s attributes:

mips_reg, IsDirty, Is32bit, NoNeedToLoadTheLo, etc.

MapRegister() sets the output attributes to xRT[]:

x86reg, Is32bit, etc.

So, after calling MapRegister, xRT->x86reg indicates which x86 register was mapped.

xRT->Is32bit

Indicates whether this register uses 32-bit logic or 64-bit logic. MapRegister maps the register accordingly. Sometimes, in an opcode sequence we will need to convert a register that is already mapped from 32bit to 64bit and vice-versa. MapRegister() handles these conversions as well.

x86reg[2].BirthDate

When we want to map a register, and all the x86 registers are already mapped, we need to flush an old register. The BirthDate field tells us which register is the oldest so that we can flush the least recently used register. Note: obviously we can't flush a register that is being used in the current instruction. ☺

Regcache.c is the heart of the register caching routines. It has functions for mapping the registers, flushing the registers, and functions for some other analysis.

How 1964 compiles constants

This section describes how 1964 combines some instructions into a single instruction. Have a look at the following sequence of MIPS instructions:

```
lui at, 0xA430
addi at, 0x000C
lw k0, [at+0x0C]
```

Since the at register contains constant data, we can simply recompile the instructions so that the result is one instruction:

```
lw k0,  [0xA430000C+0x0C]
; or simply:
lw k0, [0xA4300018]
```

So when 1964 encounters a sequence that begins with LUI, such as in the above example, it does not map these constants to registers, but instead first detects if they can be mapped as constants. Essentially, no x86 register is mapped to load a constant, but constants are stored in the compiler's ConstMap[32] array (one constant for each of the 32 MIPS GPRs). Then, when it is finally time to flush, 1964

1) Maps the constant to an x86 register.

2) Uses the flushing routine.

* With some rewiring, these 2 steps will probably become obsolete in a future build of 1964 and will store the constant to memory directly, but I'll need to determine which method is faster.

Conclusion

There are many more ways to optimize a compiler, as you can imagine. Here are a few additional concepts that extend beyond what 1964 version 0.6.0 does:

- With multiple compiler passes on a sequence of MIPS instructions, advanced analysis can be done to achieve superior optimizations. This is what is known as an "optimizing compiler".

- Regcaching can occur across multiple blocks. This means that there is no need to flush after every jump, branch, or eret. Without question, this is the most difficult thing to achieve in dynarec, but it yields the best results. One day, 1964 "may" see this.

There are other important topics worth mentioning in this document, most notably is how to handle self-modifying code. Look for these topics in a future version of this doc.

Many thanks as always to zilmar. You can find information on how to write an emulator at zilmar's site:

http://emubook.emulation64.com

Source code and binaries of 1964 are available at the 1964 official web site:

http://www.emuhq.com/1964

Thank you for reading this document. I hope that you have found it to be both educational and useful. As the 1964 dynamic recompiler progresses, so will this doc. If you have any comments, questions, or suggestions, I am available at: schibo@emuhq.com. Feedback of any kind is always helpful and much appreciated.