Dynamic Recompiler Author's Guide

Contents

This is intended to be an introduction to how to write a dynamic recompiler using the universal architecture in MAME. It is important to note that the description of how to write a dynamic recompiler below is not the only way to do so. In fact, the universal architecture is sufficiently flexible that it is possible to do something substantially different than what is described here and still make it work.

The decription below essentially describes how the first two recompilers (MIPS3 and PowerPC) operate, and provides in general a well-tested framework for future recompilers.

I suspect this article will eventually need to be broken into multiple separate pieces, but we'll keep it all together to start with.

Logic Flow

The logic flow of an interpreter is pretty easy to understand, because it works like a CPU does:

```
1.  mycpu_execute_interpreted(numcycles)
2.  {
3.     do
4.     {
5.        opcode = fetch_opcode();
6.        execute_opcode(opcode);
7.        numcycles -= opcode_num_cycles;
8.     }
9.     while (numcycles >= 0);
10. }
```

Essentially, the core execution of the interpreter consists of sitting in a loop, fetching a single opcode at a time, executing it, and counting up cycles until it is time to move on to something else in the emulation. The bulk of the code in an interpreter consists of the implementations of the individual opcodes, and must be relatively well-tuned because it is executed quite frequently.

In contrast, the logic flow of a dynamic recompiler is considerably less analagous to a CPU:

```
1.  mycpu_execute_recompiled(numcycles)
2.  {
3.     cpustate.numcycles = numcycles;
4.     do
5.     {
6.        return_code = (*recompiler_entry_point)();
7.        if (return_code == EXIT_MISSING_CODE)
8.           recompile_code_at_current_pc();
9.        else if (return_code == EXIT_FLUSH_CACHE)
10.          code_cache_flush();
11.    }
12.    while (return_code != EXIT_OUT_OF_CYCLES);
13. }
```

This may seem kind of odd at first, but let's step through the high-level flow to understand what is happening.

The first thing we do is stash the number of cycles to execute somewhere where it will be accessible to the generated code. In this case, we store it in the CPU state. Then we begin our execution loop.

Now, in an ideal world, all the code we would ever want to execute has already been translated to the target architecture, and so to execute it, we simply jump into the translated code and begin execution. If this is in fact the case, then the generated code will execute for the number of cycles we stashed in the CPU state. When it is finished, it will return to the caller with a return code of EXIT_OUT_OF_CYCLES, indicating that all the cycles have been exhausted.

Of course, this is a *dynamic* recompiler. The word dynamic here refers to the fact that we do not translate all the code up front, but rather perform translation on the fly, as new code is encountered. This means that we will sometimes attempt to execute code that hasn't been translated yet. When this occurs, the translated code will return to the caller with a special return code of EXIT_MISSING_CODE, meaning that there is no valid translated code for the current PC. In response to receiving this return code, the recompiler must translate the new code into the code cache, and then call back again to the translated code to allow it to continue processing.

It is also possible that during execution, a substantial change is made to the CPU's state, such that all previously translated code is immediately invalidated. In this situation, the translated code returns EXIT_FLUSH_CACHE, in response to which the recompiler will flush the cache. When finished, we loop back around and attempt to execute the translated code again. Of course, since we just flushed the cache, the expectation is that we will immediately return with an EXIT_MISSING_CODE so that we are forced to translate from scratch the code that was previously executing.

Setup and Initialization

Initialization of an interpreted CPU core is simple: just configure the initial state of the CPU and you're done.

With a dynamic recompiler, however, there are a number of things you need to set up and configure in order to be able to dynamically generate code on the fly. This added complexity means that initialization of the CPU state itself gets buried under the details of the recompiler. For this reason, it is recommended that you create a separate "common" module which contains the actual CPU state and functions you can call to initialize and reset the state, as well as other functions that would be shared between a recompiler and an interpreter. This also means that if you decide to maintain an interpreter alongside your recompiler, some of the interpreter logic can be replaced by calls into this common code, ensuring that complicated behaviors are consistent between the two.

To explain what needs to be done during recompiler initialization, this article will walk through the MIPS3 recompiler initialization code step-by-step, explaining along the way what each part of the function is doing and why it is necessary. First, though, I want to mention one global:

```
1.  static mips3_state *mips3;
```

The first thing to point out is that this global mips3 variable contains a pointer to the MIPS3 CPU state. The actual state structure (mips3_state) is defined in the common module which was described above. Note that unlike an interpreter, where the CPU state is typically accessed via globals for speed, for a recompiler you pretty much need to use a pointer to memory that has been specially allocated from the code cache, in order to enable your generated code quick access to the CPU state. In terms of speed, there is no benefit to a pointer versus globals, so this is not a difficult choice to make.

Now, getting into the actual initialization function:

```
1.  /* allocate enough space for the cache and the core */
2.  cache = drccache_alloc(CACHE_SIZE + sizeof(*mips3));
3.  if (cache == NULL)
4.      fatalerror("Unable to allocate cache of size %d", (UINT32)(CACHE_SI
```

We first allocate our code cache. There is a helper module drccache.c which contains some basic functions for allocating and managing space within the code cache. Note that CACHE_SIZE is hard-coded (in this case, it is 32MB). This value should be large enough to ensure that you don't need to frequently flush the cache and throw away everything you've generated. In addition to the cache itself, we also allocate space for the mips3_state.

```
1.  /* allocate the core memory */
2.  mips3 = drccache_memory_alloc_near(cache, sizeof(*mips3));
3.  memset(mips3, 0, sizeof(*mips3));
```

Once we have the cache, we allocate memory from it for the mips3_state. Note that we allocate this memory from the "near cache", where it can be directly accessed by the generated code.

```
1.  /* initialize the core */
2.  mips3com_init(mips3, flavor, bigendian, index, clock, config, irqcallba
```

After allocating the cache, we call through to the common module's initialization function (all MIPS3 common module functions are prefixed with mips3com_), which will configure the mips3_state to the initial processor state, configure save states for the processor, and do any initial configuration.

```
1.  /* allocate the implementation-specific state from the full cache */
2.  mips3->impstate = drccache_memory_alloc_near(cache, sizeof(*mips3->imps
3.  memset(mips3->impstate, 0, sizeof(*mips3->impstate));
4.  mips3->impstate->cache = cache;
```

Now that the core is allocated and initialized, we allocate more "near cache" memory for the implementation-specific state. This is a structure defined at the top of the module which contains all of the recompiler-specific state that the common module has no knowledge of. Conveniently, the shared mips3_state contains a pointer member *impstate* which is provided specifically for the purpose of storing a pointer to our implementation-specific state. One of the most important bits of implementation-specific state is a pointer to the cache object, so we immediately store that.

```
1.  UINT32 flags = 0;
2.
3.  /* initialize the UML generator */
4.  if (FORCE_C_BACKEND)
5.      flags |= DRCUML_OPTION_USE_C;
6.  if (LOG_UML)
7.      flags |= DRCUML_OPTION_LOG_UML;
8.  if (LOG_NATIVE)
9.      flags |= DRCUML_OPTION_LOG_NATIVE;
10.
11. mips3->impstate->drcuml = drcuml_alloc(cache, flags, 8, 32, 2);
12. if (mips3->impstate->drcuml == NULL)
13.     fatalerror("Error initializing the UML");
```

At this point, we have allocated all our necessary state, so we are ready to create a new UML object, which is what allows us to generate universal machine language code on the fly. There are 5 parameters to this function, and understanding their purpose and meaning is crucial, so let's examine them one by one:

- *cache* is the first parameter, and it is simply a pointer to the code cache object we allocated; the UML code is specifically permitted to make use of the cache for its own needs (allocating memory, etc), and will ultimately provide the same pointer to the back-end
- *flags* is a set of bit flags ORed together that control the UML's behavior:
  - the DRCUML_OPTION_USE_C flag tells the UML to ignore any native back-ends that might exist and instead always use the C back-end; this is useful for debugging issues in the back-ends, so you can compare behaviors against the C back-end
  - the DRCUML_OPTION_LOG_UML flag instructs the UML to dump a disassembly of each block of UML that is generated; this disassembly is written to the file **drcuml.asm** in the same directory as the MAME application
  - the DRCUML_OPTION_LOG_NATIVE flag is passed down by the UML to the back-end, and tells the back-end to generate a disassembly of each block of native code generated; the name of this file is back-end specific
- *modes* is the third parameter (in this case 8); the concept of modes will be explained a bit farther on
- *addrbits* is the fourth parameter (in this case 32) and specifies the number of valid address bus bits to be used for looking up the current PC; note that this value should reflect the number of *logical* address bits, since the PC is looked up based on its logical address
- *ignorebits* is the fifth parameter (in this case 2) which specifies the number of low order bits to ignore in the current PC; in the MIPS case, for example, each instruction is exactly 4 bytes long and starts/ends on an instruction boundary; this means that the low 2 bits can be ignored when computing a unique index to look up

Given this information, the UML can allocate itself and the back-end generator, and knows how to size all of its internal structures.

```
1.  /* add symbols for our stuff */
2.  drcuml_symbol_add(mips3->impstate->drcuml, &mips3->pc, sizeof(mips3->pc
3.  drcuml_symbol_add(mips3->impstate->drcuml, &mips3->icount, sizeof(mips3
4.  /* more deleted */
5.  drcuml_symbol_add(mips3->impstate->drcuml, &mips3->impstate->mode, size
```

Now that the UML is created, we can improve the output of our disassembly by providing symbols. For each important symbol that might be referenced by the code, its address and size are provided along with a friendly string name. Only a partial excerpt is included here, as there are quite a large number of symbols that the MIPS recompiler registers. The symbols can live anywhere, so feel free to provide as much detail as you'd like when adding symbol values.

```
1.  drcfe_config feconfig =
2.  {
3.      COMPILE_BACKWARDS_BYTES,     /* code window start offset = startpc
4.      COMPILE_FORWARDS_BYTES,      /* code window end offset = startpc +
5.      COMPILE_MAX_SEQUENCE,        /* maximum instructions to include in
6.      mips3fe_describe             /* callback to describe a single instr
7.  };
8.
9.  /* initialize the front-end helper */
10. if (SINGLE_INSTRUCTION_MODE)
11.     feconfig.max_sequence = 1;
12. mips3->impstate->drcfe = drcfe_init(&feconfig, mips3);
```

The next step is to initialize the front-end. In brief, the front-end (which is described in more detail in the next section) is responsible for walking through the code, determining register dependencies and code flow, and for breaking the code up into blocks to be recompiled. The drcfe_config structure describes the parameters for the code walk and provides a callback function pointer to the architecture-specific opcode analyzer function.

```
1.  /* copy tables to the implementation-specific state */
2.  memcpy(mips3->impstate->fpmode, fpmode_source, sizeof(fpmode_source));
3.
4.  /* compute the register parameters */
5.  for (regnum = 0; regnum < 34; regnum++)
6.  {
7.      mips3->impstate->regmap[regnum].type = (regnum == 0) ? DRCUML_PTYPE
8.      mips3->impstate->regmap[regnum].value = (regnum == 0) ? 0 : (FPTR)&
9.      mips3->impstate->regmaplo[regnum].type = (regnum == 0) ? DRCUML_PTY
10.     mips3->impstate->regmaplo[regnum].value = (regnum == 0) ? 0 : (FPTR
11. }
12.
```

Now we're in the home stretch. If there is any required static initialization of the implementation-specific state, it can be done now. In this case, we have a table that we copy so that it is always accessible from the near cache. And then we build up a table of pointers to each of the registers, which we will use

during code generation to quickly determine the parameter type and value given a MIPS register number. Note that because MIPS register 0 is always hard-coded to 0, we specify the type to be DRCUML_PTYPE_IMMEDIATE instead of DRCUML_PTYPE_MEMORY. This is one of the reasons why we store both a type and value for each parameter.

```
1.  /* mark the cache dirty so it is updated on next execute */
2.  mips3->impstate->cache_dirty = TRUE;
```

Finally, we mark cache dirty, which will force it to be flushed on the next (first) call to the CPU's execute function.

Front-end Analysis

Once it has been determined that there is no translation present for a given PC, the dynamic recompiler is responsible for examining the emulated code and generating the equivalent UML output. The tricky part, believe it or not, is the "examining the emulated code" bit. For maximal efficiency, the recompiler is going to want to translate more than just one instruction at a time; however, it doesn't want to waste a bunch of time translating instructions that aren't going to be executed.

Because this is a common challenge for any recompiler, the universal DRC provides a generic front-end to make it easier. Before proceeding, it is important to understand a few basic terms:

- A code **window** is a predefined PC-relative lower and upper bound within which the analysis will take place. An example window from the MIPS3 case has a minimum of -128 and a maximum of 512. This means that as soon as any path through the code goes more than 128 bytes before the starting PC, or more than 512 bytes beyond the starting PC, it effectively stops the analysis.
- A code **block** is the collection of instructions that result from walking through the code from the starting PC until all known paths terminate (either explicitly or by branching outside of the window)
- A code **sequence** is a subset of a code block where all instructions from beginning to end are guaranteed to execute sequentially, with no branches into the middle
- An opcode **descriptor** is a small structure that provides information about a single emulated instruction

To illustrate these concepts, here's some made-up MIPS assembly:

```
 1.  BFBFFFFC: ...
 2.  BFC00000: ori  r2,0,$1000   ; block seq1
 3.  BFC00004: ori  r3,0,$8000   ; block seq1
 4.  BFC00008: sw   0,0(r2)      ; block      seq2
 5.  BFC0000C: addi r3,r3,-1     ; block      seq2
 6.  BFC00010: bgez r3,BFC00008  ; block      seq2
 7.  BFC00014: addi r2,r2,4      ; block      seq2 seq3
 8.  BFC00018: jr   r31          ; block           seq3
 9.  BFC0001C: nop               ; block
10.  BFC00020: ...
```

Now let's say our code cache is empty and we begin executing. Immediately, we find out that PC=BFC00000 is not present in the cache, so we exit and perform a recompilation. As a first step, we call the front-end, which we initialized previously to describe our window size and a pointer to our architecture-specific opcode analyzer.

At this point, the front-end's job is to walk through the code to identify the largest block that fits within the window, and to identify sequences within the block. To do this, it starts with the first opcode and calls the architecture-specific analyzer with an empty descriptor. The analyzer's job is to set various flags in the descriptor to describe the instruction. Upon return, the front-end has enough information to know how to proceed in its analysis. It continues doing this until it has exhausted all code paths.

Let's have a look at the opcode descriptor:

```
1.  <insert structure here>
```

<Describe fields>

<Walk through example above>

Static Subroutines

Common Generation Functions

General Opcode Behaviors

Specific Opcode Behaviors