Porting QEMU to Plan 9: Strategy

Nathaniel Wesley Filardo

August 8, 2007

Abstract

This paper discusses the difficulties which must be overcome to port the QEMU processor emulator [1] to Plan 9. It begins with a detailed look at QEMU's internal machinery and outlines the difficulties encountered. For each, it discusses the currently favored approach towards a workable solution. In many cases, alternatives and mechanisms for subsequent improvements and optimizations are also presented. It is hoped that this paper also provides useful groundwork for other, future ports of QEMU to novel platforms and compilers.

Contents

1 Overview

Section contract: introduce QEMU, set stage for horror of section 2. In particular, I don't even want to talk about translation in any real way up here.

QEMU is a fast, portable emulator of many architectures, including IA-32 (X86), PPC, and SPARC. It seems worthwhile to bring it to the Plan 9 environment so that certain Linux applications (*e.g.*, firefox) can be made available and to provide an environment for kernel testing.

1.1 The Nature of QEMU

Bochs [3] is a well-known, portable IA-32 emulator. Bochs uses simulation to emulate the guest system: it disassembles and simulates each instruction one at a time. While straightforward and as accurate as desired by the implementors, this approach is slow as many host instructions (function dispatch, entry, evaluation, and return) must be run for each guest instruction.

QEMU, on the other hand, uses an on-the-fly translation technique where guest code is first translated into an equivalent series of so-called "micro-operations," which are then *copied*, *modified*, and *concatenated* to produce a block of native code . These micro-ops range in complexity from simple simulated register transfers to integer and floating point math to memory load and store operations (which require simulating the guest architecture's paging mechanism). Translation is currently done per basic block of guest code and translations are cached for reuse. In the absence of interrupts, translations are chained together so that control flow stays within guest code. Interrupts arrive asynchronously and undo these chains so that control flow returns to the main QEMU code.

As an explicit comparison, it will be useful to use the X86 `ARPL` instruction[1]. The Intel "type" of this function is `ARPL r/m16,r16`, meaning that it takes two operands, the first of which is a 16 bit register or memory location and the second is a 16 bit register. Bochs will decode enough of the instruction to determine that it is some form of `ARPL` and will call `BX_CPU_C::ARPL_EwGw`, a function of 55 lines which contains calls to determine the type of operands, conditional branches, and calls to load and store functions. QEMU, on the other hand, will decode the instruction once into a specialized series of micro-ops:

- Move the first operand into the 0^{th} internal simulation register. Depending on the bits of the "ModR/M"[2] byte following the ARPL instruction byte, the translator will pick either a memory fetch or a register transfer; the test is performed by the translator, so the generated instruction stream will not include a branch.
- Fetch a 16-bit value from the host state variable which holds the indicated guest CPU register *or* emulate a fetch from memory and place it in QEMU's 1^{st} internal simulation register.
- Do the core of the `ARPL` instruction. This micro-op demands that its operands are in temporary registers 0 and 1, as has just been set up.
- Move the result from the 0^{th} temporary back to the original host state variable holding the indicated register or emulate a store to memory. Again, this will be specialized *either* to a store or a register move.

Once the sequence of micro-ops for a basic block has been determined, the translator then converts the sequence into host machine code and stores the result in the translation cache for subsequent use.

The core concept of QEMU, ignoring many fancy tricks and optimizations, can then be understood in terms of a loop with this basic structure:

1. If a guest interrupt is pending, adjust the emulated machine state as appropriate to invoke the guest's interrupt handler.
2. If the translation cache does not contain a translation of the basic block starting with the next instruction we wish to execute, perform the translation and store it in the cache.
3. Perform a subroutine call into the basic-block translation starting at the next instruction

The result is an execution pattern alternating between bursts of host instructions implementing one basic block of guest instructions and a sequence of host instructions performing house-keeping and translation.

1.2 Simulated Hardware

Both Bochs and QEMU simulate hardware at a very low level. Both have software representations of buses and peripherals such as video and network cards and disk controllers. Both Bochs and QEMU provide to the simulation accurate models of a limited set of hardware, including interrupt controllers, bus drivers, disk controllers, disk drives, keyboards, mice, video cards, and network cards. Over time, this set has grown to include a reasonable selection of devices likely to be supported by guest operating systems. Both Bochs and QEMU use BIOSes run inside the simulation

[1] An odd instruction related to segment selector requested privilege level.

[2] For details, the masochistic should consult Intel's Instruction Set Reference. Not suitable for all audiences.

to initialize certain parts of the hardware, a design decision which allows the device emulators to remain faithful to the original hardware.

Outside the simulation, these device drivers (drivees?) make use of host features to provide the simulation and the user with desired functionality. Some examples are

- Video frame-buffers are exposed via the user's choice of UIs. For QEMU, options include a SDL window, a VNC server, and no graphical output.
- Networks under QEMU can be disabled, bridged to the host kernel, created over a UNIX socket using a virtual Ethernet protocol (allowing other QEMU and compatible simulators and virtualizers to participate), or entirely simulated by QEMU.

1.3 Portability

QEMU is reasonably modular, with guest-specific parts of the emulator largely factored out into their own files and directories[3]. All guests speak the same interface to the core, drivers, and dynamic translator. Of the some 111,000 lines of code[4] for the entire QEMU system, the guest-specific components constitute roughly a third. The X86 guest in particular occupies under 8,000 lines of code. The relative compactness of the guest descriptions enables QEMU, unlike Bochs, to simulate a large number of guests[5].

QEMU requires that its platforms expose information about their compiled executables, for the dynamic translator's use. Fortunately, most of this information is desirable for more "respectable" reasons such as debuggers, dynamic loaders, or support of separate compilation. Most modern platforms offer most of the needed infrastructure or are a small patch away from doing so.

Further, QEMU is written almost entirely in C, creating a layer of isolation between host and guest environments. Notably the dynamic translator is written entirely in C with some GNU extensions. This structural portability, coupled with GCC's large list of supported systems, endows QEMU with a large degree of portability across host systems. The obstacles for porting to Plan 9 will be largely the use of GCC extensions, teaching Dyngen about Plan 9's a.out format, and some UNIXisms in the host driver code.

1.4 Roadmap

The next section gives an in-depth look at the current state of QEMU's universe, focusing on the dynamic translator as both a provider of tools to other parts of QEMU and as a consumer of the larger system's offered tools. Then, with that groundwork, we discuss mechanisms for offering the same tools on a Plan 9 system.

FIXME

2 QEMU The Emulator

Section contract: We narrow our discussion to the heart of QEMU: the dynamic translator. For the moment, we neglect interactions with the user, other programs, and hardware. The goal here is to explain everything from the perspective of data structures, operators, and application thereof. The reader should come out of this section with a good understanding of the dynamic translator and micro-ops library as consumers and producers of tools, not necessarily with the implementation details.

QEMU uses a portable dynamic code translator [1] to achieve fast emulation of guest code. It achieves its high degree of portability relative to other code generators by being implemented largely in C with some GNU extensions. It does not natively know the instructions of its host architecture – instead, each guest specifies, in C, a library of micro-operations as well as a guest-code disassembler and translator into its micro-ops vocabulary. These micro-operations can be thought of as a kind of virtual machine, albeit one optimized for simulation of the guest system. The operations themselves include register transfer, explicit (rather than implicit, see [1]) condition code update code, bitwise operations, integer and floating math, and memory load and store operations.

2.1 Advanced Tricks With Translation Buffers

QEMU improves performance beyond the basic approach outlined in Section 1.1 in a few key ways. First, large or expensive host operations (such as emulating guest MMU operations) are not directly placed into the translation buffer, but are contained in helper functions called from the micro-ops. Second, an escape hatch mechanism is provided for synchronous faults that occur in the middle of a translation block. Third, optimizations are performed to improve the efficiency of the instruction sequences emitted by the translator. Forth, control flow is complicated in ways which allow

[3]Some per-host and per-target code remains in common files, using `#ifdef` to select the right one

[4]grep -c \;

[5]Of interest in particular to the Plan 9 community are its X86, ALPHA, MIPS, PPC, and possibly SPARC targets.

```
l1:
  ADDL $0x2, %EBX
  DEC %EAX
  JNZ l1
```

(a) A host instruction stream

```
/* ADDL $0x2, %EBX */
env->EBX += 2;
compute_cc_z(env->EBX);
compute_cc_c_addl(env->EBX, 2);
env->EIP += 3;

/* DEC %EAX */
env->EAX -= 1;
compute_cc_z(env->EAX);
env->EIP += 3;

/* JNZ l1 */
if(env->CC_Z != 0)
  env->EIP = l1;
else
  env->EIP += 6;

return;
```

(b) A naïve translation showing condition code and EIP updates.

```
/* ADDL $0x2, %EBX */
env->EBX += 2;
compute_cc_c_addl(env->EBX, 2);

/* DEC %EAX */
env->EAX -= 1;
compute_cc_z(env->EAX);

/* JNZ l1 */
if(env->CC_Z != 0)
  env->EIP = l1;
else
  env->EIP += 12;

return;
```

(c) A translation without unnecessary computation.

Figure 1: A simplified view of translation into host code, showing optimizations on condition codes and instruction pointer calculations. Translations are represented in C rather than micro-op names for clarity.

the execution of many basic blocks worth of code between invocations of the house-keeping, decoding, and translation code paths.

2.1.1 Function Calls To Reduce Translation Size

The X86 instruction `CPUID` is remarkably complex[6] and requires about 75 lines of C even in QEMU's simplistic implementation[7]. Instead of the usual approach, where the translator would copy code to implement the `CPUID` instruction into the translation buffer, in this case it generates a call to a non-specialized helper function. Rather than special-case instructions with helper function implementations, micro-ops containing function calls are defined for these instructions (*e.g.*, the micro-op selected for `CPUID` consists only of a call to the helper function, `helper_cpuid`). Function calls are also notably used to emulate guest MMU accesses – MMUs are generally complex and QEMU uses a MMU translation cache to speed up MMU emulation. Placing all of this complexity in each translation buffer at each memory operation site would be extremely expensive.

2.1.2 Synchronous Fault Escape Hatch

A complication induced by using translation buffers rather than instruction-by-instruction simulation is that an exception, such as an MMU fault, might arise in the middle of a buffer. QEMU has a mechanism, based on `longjmp()`, to bail out from a translation buffer back to the emulation core if a fault synchronous to the instruction stream must be issued. Upon dealing with the synchronous fault, a new translation buffer is created starting from the interrupted instruction[8].

2.1.3 Lazy Evaluation

Every instruction implicitly modifies the instruction pointer and almost every instruction makes updates to the processor's condition codes (zero, overflow/carry, etc.). However, it is relatively rare that the guest code truly cares about its instruction pointer's value, as long as the instructions are dispatched in the right order. The condition codes are also rarely looked at, usually by conditional jumps.

Since an entire basic block of code is translated, QEMU avoids updating the instruction pointer until the block is finished or explicitly reads it. Similarly, QEMU targets carry out liveness analysis passes over the condition codes and substitute in "simplified" versions of the translation to avoid unnecessary computation. Consider, as an explicit

[6]Doubtless introduced with only the best of intentions, it is now a fossil record of the evolution of the X86 architecture.

[7]The QEMU implementation is a switch statement which loads hard-coded values into CPU registers. Since QEMU does not provide the ability to switch on and off individual CPU features and since any (reasonable) answer it provides for things like cache sizes is as good as another, this is a reasonable approach.

[8]Execution cannot resume in the middle of that translation buffer as it may be gone from cache.

example, the loop in Figure 2.1.3, running on a(n extremely simplified) X86. In addition to the arithmetic manipulation, the processor specification requires that

- `ADDL` must update both the zero and carry condition codes.
- `DEC` must update the zero condition code.
- Both instructions must move the instruction pointer forward to the next instruction.

We can see immediately that `ADDL`'s updates to the zero flag are invisible to the guest: the next instruction clobbers it. Since neither the `ADDL` nor `DEC` read the instruction pointer, the translator can also eliminate two updates to the instruction pointer. Here, then, we can simply update the instruction pointer once, at the end of the basic block. However, as the next instruction after the `JNZ` (in the not-taken path) might be anything, we must still have `ADDL` compute the carry flag[9].

This lazy evaluation interacts with the synchronous fault mechanism. Upon a synchronous fault, the emulator core forces evaluation of the instruction pointer and condition codes so that it may emulate the guest architecture's interrupt mechanism. This is similar to what is done on superscalar CPUs.

2.1.4 Chaining Translation Buffers

The use of translation buffers reduces the per-instruction overhead dramatically but is, in a sense, rather unsatisfactory. Control flow must return to the emulator's main loop every time, even if the basic block ends with a jump to a fixed address (as is common, for example, with basic blocks resulting from if-then-else or switch-case control flow). In the limit, we would like to translate the entire program at once. However, this likely wouldn't fit in cache and we'd need a mechanism to exit an infinite loop once we'd called into one.

Instead, we can still translate on the level of basic blocks (which allows the optimizations above) but construct a mechanism for chaining them together, so that control flow does not always need to first return to the emulator loop. This can be achieved with a pattern of specialized micro-ops implementing the following algorithm:

1. If the successor translation block is known, jump to it. Otherwise, do nothing. QEMU calls this operation `GOTO_TB()`, but from the translator's perspective it is encapsulated in micro-op bodies.
2. Set the guest's program counter to the next instruction we wish to execute after leaving this basic block.
3. Return to the emulator's loop. Some housekeeping is required here, but the ultimate operation is called `EXIT_TB()`, which is also provided to the translator by a micro-op.

When the main loop is invoked because no successor was defined, there is an opportunity to look up the specified program counter in the translation cache, obtain the address of the associated translation buffer, and patch the current translation buffer so that it will know how to jump directly to its successor next time.

This also handles the infinite loop case neatly. If QEMU enters a sequence of translation buffers chained together in an infinite loop, eventually a SIGALARM will force QEMU into a signal handler. It can then *undefine* the successor values of recently executed translation buffers and return. This will cause execution to "fall out" into the house-keeping main loop "soon," which will then force the guest into its timer interrupt handler.

2.1.5 Jumping Around Translation Buffers

The above mechanism for chaining works as long as each basic block has only one successor. However, conditional jumps have two successor blocks: the one taken on condition match and the other where the conditional jump acts like a no-op. We will call these the "branch taken" and "branch not taken" successors, respectively.

One simple approach would be to prohibit chaining translation buffers ending on conditional branches. Then, the conditional branch micro-ops would simply set the guest instruction pointer depending on their tests. Control flow would return to the emulator loop and the right thing would happen. However, this requires that every conditional jump cause a return to the emulator loop, every time through – that is, even after the successor for a given path has been translated and placed in cache.

In order to take advantage of translation block chaining in the presence of conditional jumps, there must be (at least[10]) two sets of chaining meta-data, one for each arm of the jump. `GOTO_TB()` must be parameterized to specify which set of chaining data is to be considered for this trip out of the translation buffer. In the case where `GOTO_TB()` does not know its successor, the emulator loop needs to be told which chaining meta-data is to be updated.

These changes in and of themselves are insufficient: despite parameterization, we have not provided a way for the translation buffer to conditionally call one path or the other. QEMU's micro-ops libraries provide micro-ops that transfer control to another point by jumping, using a mechanism QEMU calls `GOTO_LABEL_PARAM()`, which may be thought of

[9] A sufficiently advanced optimizer might move the carry flag update into the "else" branch of the `JNZ`. Such is orthogonal to the mechanisms of QEMU, so we need not consider it now.

[10] One could imagine a more sophisticated translator that could place several conditional jumps into one translation unit so long as there were not possibility of looping.

as simply a host `JMP` instruction with enough room in the instruction to jump to any address. As with `GOTO_TB()` and `EXIT_TB()`, `GOTO_LABEL_PARAM()` is exposed to the translator by being contained within specialized micro-ops.

Recall that the translation procedure involves disassembling guest instructions, selecting an appropriate sequence of micro-ops, and then translating that sequence into host instructions. During the selection phase, the translator can look up the size of each micro-op's host code. Thus it can readily learn the which offsets into the translation buffer are between micro-ops. It can even bind these offsets to named "labels" during translation and pass them to the code generator.

This somewhat tortured mechanism allows us to translate guest conditional jumps into host condition testing and unconditional jumps. Specifically, a guest conditional jump (which will be at the end of a translation buffer) now looks, in schematic, like:

```
  if(condition)
    GOTO_LABEL_PARAM(l1)
  GOTO_TB(nottaken)
  Write nottaken successor address to the guest instruction pointer
  Write "nottaken successor" for the emulation loop
  GOTO_LABEL_PARAM(l2)
l1:
  GOTO_TB(taken)
  Write taken successor information to the guest instruction pointer
  Write "taken successor" for the emulation loop
l2:
  EXIT_TB()
```

That is, if the condition is true, we will follow the branch taken successor branch (the code after `l1`) and the first time through will return to the emulator loop and get patched up for next time. If, next time through, the condition is still true, we will directly chain to the appropriate translation block. If not, we will proceed to the branch not taken successor and the same thing will happen.

2.1.6 Translation Buffers as Data Structures

Translation buffers (and the micro-ops that comprise them) may be thought of as data structures, generated and consumed throughout QEMU's execution, supporting unusual operations once constructed:

- Set/Reset the branch not taken / sole successor.
- Set/Reset the branch taken successor, if this block ends on a conditional branch.
- Execute the translation buffer.
- Bail out of execution from the middle of the translation buffer.

2.2 A Closer Look at Micro-Ops

The micro-ops are written in (GNU) C, but manipulated as largely opaque binary data (once compiled) by the dynamic translator. That is, the dynamic translator uses as little introspection into the micro-ops compiled form as it can get away with; in particular, it does not attempt to disassemble the generated instruction stream. There are, however, some "charming" features of the dynamic translator's use of these compiled functions.

2.2.1 Coring

Since C functions complete by returning, and most compilers produce code with prologues on entry and epilogues to return, the existing dyngen design is to mechanically strip both to extract the core of the function. These cores can then be pasted together and the entire mass given a prologue and epilogue to create a function at runtime. Assuming that each core has a unique return point in its epilogue (that is, at its highest address), this will work exactly as desired: instead of returning, each concatenated function will simply hand control off to the next by falling through.

2.2.2 Register Allocation

GCC has extended the C language to allow some control over the register allocator. QEMU's targets, whenever possible, assign host registers to hold a pointer to the emulator environment, the temporary registers, and a subset of the guest's registers. The goal is to reduce memory traffic and address computations for the common cases seen in the micro-ops library.

However, as some hosts may have registers smaller than some guests, most guests provide code for the case where registers are unavailable. This is fortunate, as it means platforms whose C compilers do not allow such fine-grained (and non-portable) control over the compiler's output can use these other pathways for their initial port. Section 4.3 discusses a mechanism for using register allocation on Plan 9 in more detail.

2.2.3 External References

Almost all micro-ops reference global variables[11], and some make function calls to helper functions. Examples of helpers notably include complex things such as MMU emulation and odd things like the X86 instruction `CPUID`'s implementation. Thus, whenever a micro-op core is copied around, it needs to be rewritten using relocation information so that these references are still valid. While the linker does, indeed, do this, the function bodies are dynamically copied into translation units at runtime, meaning that the relocation pass must be done within QEMU itself.

2.2.4 Constant Folding

An additional use of relocation meta-data is to emulate guest operations with immediate data (*e.g.*, constants to be loaded into registers). Consider that QEMU might be asked to simulate both `movl $0x5, %eax` and `movl $0x2BADD00D, %eax`. Possible approaches to this problem include having specialized micro-ops (perhaps the ability to load, byte-wise, into `T0`), tabulating constants and emitting memory-to-register transfer instructions using much the same approach as above, or perhaps (ab)using the stack to pass parameters to translation units. However, it would be ideal if guest-code immediate values could, whenever possible, be host-code immediate values as well. For example, upon encountering the X86-32 operation `movl $0x5, %eax` while running on an X86-64, we would like to emit `movl $0x5, %r15d` into the translation buffer[12].

Constants may be folded into the instruction stream by manipulating the relocation of specially named global variables. To separate this use of relocation from the more standard relocation done for functions, we term this "abusive relocation." Details may be found in Section 3.4.2.

2.2.5 Providing Non-local Control Flow

As discussed above, there are three mechanisms provided to translation buffers by micro-ops for non-local control flow:

- `EXIT_TB()` for returning to the emulator loop,
- `GOTO_LABEL_PARAM()` for branching within a translation buffer, and
- `GOTO_TB()` for chaining translation buffers.

`EXIT_TB()` is relatively uninteresting, so we focus here on the other two.

Much as we could refer to "left" and "right" successors of translation blocks, we may refer to "left" and "right" successors of individual micro-ops. All micro-ops (except those which exit the translation buffer) have a natural "left" successor of the next micro-op in the translation buffer. Micro-ops involved in non-local control flow may additionally have a "right" successor, which may be said to be either "near" if it is in the current translation buffer and "far" if it is in another translation buffer. For simplicity (despite the *capability* of the mechanisms to allow this), QEMU does not jump into non-zero offsets of other translation buffers.

`GOTO_LABEL_PARAM()` is implemented by handing control flow (via inline assembler) to a special class of abusive symbol. For each instance of `GOTO_LABEL_PARAM(label)`, the code generator calculates the actual host instruction address corresponding to `label`. It then abusively sets the abusive symbol's value to this address when copying the micro-op body. Notice that `GOTO_LABEL_PARAM()` *always* dispatches control to the near, right successor of this micro-op. However, micro-ops may have conditionals around their use of `GOTO_LABEL_PARAM()`, as was seen in the example of translation buffers having multiple successors.

For `GOTO_TB()`, the situation is more complicated, as there are three separate mechanisms for implementing `GOTO_TB()`. The simplest makes use of a GNU C extension of goto and emits code which reads an address from the translation buffer meta-data and jumps there. The code generator *exports* the address of the instruction after the jump to the emulator. After code generation, but before running the translation, the emulator resets the recorded address to the next instruction. This results in `GOTO_TB()` being an expensive no-op instruction. Later, when the emulator learns the address of the next translation buffer, it will store it back into the appropriate slot in the previous translation buffer's meta-data.

2.2.6 Micro-Ops as Data Structures

Much as we could view translation buffers as odd data structures, micro-ops themselves are similarly odd data structures, supporting a larger vocabulary of mind-bending operations statically, at run time; dynamically, at the request of the emulator; and at execution time.

- Execution Operations Within A Micro-Op:
  - Make a function call.
  - Read from global store (and/or registers).
  - Write to global store (and/or registers).

[11] Especially in absense of QEMU's host-register allocation.

[12] Register %r15d is used on an X86-64 host to store the X86 guest %EAX register.

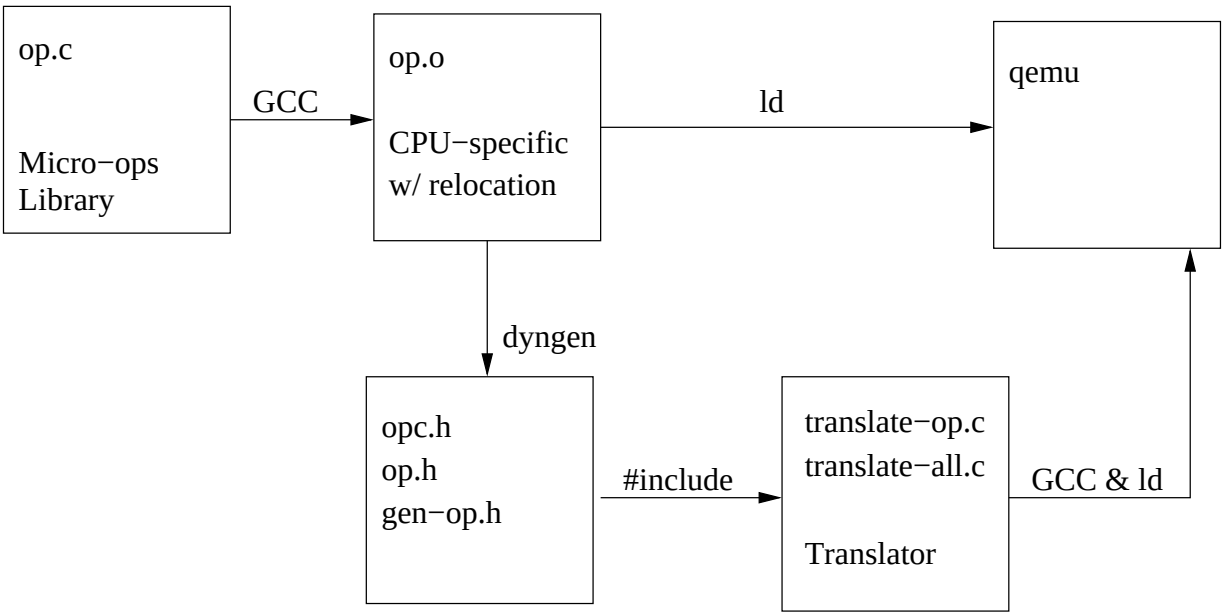


Figure 2: Compiling the micro-ops library.

  - Go to the next micro-op[13] ("branch not taken successor")
  - Transfer control to another micro-op in this translation buffer("near branch taken successor").
  - Transfer control to another micro-op in another translation buffer("far branch taken successor").
  - Exit this translation buffer.
- Static Operations:
  - Extract core.
  - Enumerate relocation requirements.
  - Enumerate constant-folding parameters.
  - Enumerate control-flow parameters and exports.
- Dynamic Operations At Translation Time:
  - Copy core to target address.
  - Relocate function calls.
  - Fold a value in for a constant parameter.
  - Set a jump's destination label.
  - Export `GOTO_TB()` destination patch points.
  - Export `GOTO_TB()` labels.
- Dynamic Operations After Translation Time:
  - Set a `GOTO_TB()` destination.

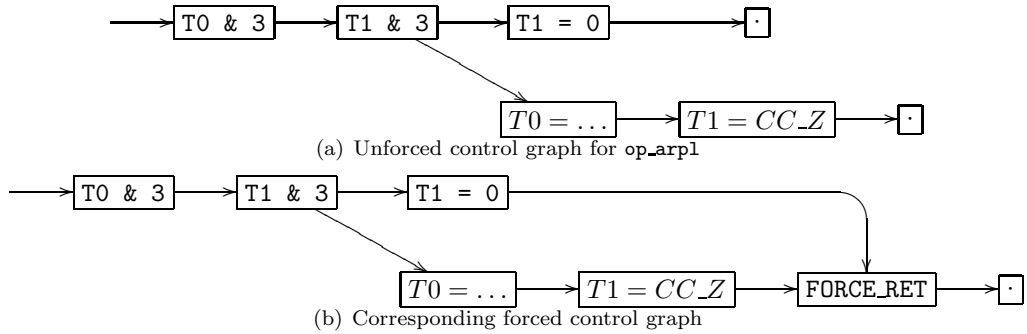
We now turn our attention to the implementation details enabling each of these operations, discussing QEMU's current implementation on GCC. In the next section, we will give a parallel structure for work on Plan 9.

3 Achieving Dynamic Translation on UNIX/GCC Hosts

3.1 Compiling the Dynamic Translator

The compilation phases are shown in Figure 2. A tool called "dyngen" takes the compiler-generated host-specific version of the micro-op library and produces C necessary for the runtime translator to make use of these routines. The exact details of processing are deferred briefly.

[13] This is not nearly as trivial as one might wish.



(a) Unforced control graph for `op_arpl`

(b) Corresponding forced control graph

Figure 3: Forcing to an "obviously serializable" control flow graph.

```
000000000049e943 <op_arpl>:
  49e943:  mov    %r15d,%edx
  49e946:  mov    %r12d,%eax
  49e949:  and    $0x3,%edx
  49e94c:  and    $0x3,%eax
  49e94f:  cmp    %eax,%edx
  49e951:  jae    49e96c <op_arpl+0x29>
  49e953:  mov    %r15d,%edx
  49e956:  mov    %r12d,%eax
  49e959:  mov    $0x40,%r12d
  49e95f:  and    $0xfffffffffffffffc,%edx
  49e962:  and    $0x3,%eax
  49e965:  mov    %edx,%r15d
  49e968:  or     %eax,%r15d
  49e96b:  retq
  49e96c:  xor    %r12d,%r12d
  49e96f:  retq
```

(a) Without the use of `FORCE_RET()`.

```
000000000049e943 <op_arpl>:
  49e943:  mov    %r15d,%edx
  49e946:  mov    %r12d,%eax
  49e949:  and    $0x3,%edx
  49e94c:  and    $0x3,%eax
  49e94f:  cmp    %eax,%edx
  49e951:  jae    49e96d <op_arpl+0x2a>
  49e953:  mov    %r15d,%edx
  49e956:  mov    %r12d,%eax
  49e959:  mov    $0x40,%r12d
  49e95f:  and    $0xfffffffffffffffc,%edx
  49e962:  and    $0x3,%eax
  49e965:  mov    %edx,%r15d
  49e968:  or     %eax,%r15d
  49e96b:  jmp    49e970 <op_arpl+0x2d>
  49e96d:  xor    %r12d,%r12d
  49e970:  retq
```

(b) With use of `FORCE_RET()`.

Figure 4: GCC 4.1.3 on X86-64 compilation results for `op_arpl`.

3.2 Requirements of Micro-Op Control-flow Graphs

In order to support the current "coring" mechanism, micro-ops must compile into assembler such that they have a unique epilogue that happens to be at the highest address. Some micro-ops, though, are sufficiently complex that it is not obvious (or necessary) that all paths converge on the function's unique epilogue. Consider the X86 guest's micro-op that is the core of the X86 `ARPL` instruction (from `target-i386/op.c:1180`):

```
void OPPROTO op_arpl(void)
{
    if ((T0 & 3) < (T1 & 3)) {
        T0 = (T0 & ~3) | (T1 & 3);
        T1 = CC_Z;
    } else {
        T1 = 0;
    }
    FORCE_RET();
}
```

`Tn` and `CC_Z` are preprocessor macros for the temporary registers used by the translation and for the condition code zero flag, respectively. In the absence of the mysterious `FORCE_RET()`, the control flow graph is as shown in Figure 3(a). In this case, there may well be a "return" instruction in the middle of the host instruction stream, rendering the micro-op unsuitable for concatenation[14], as in Figure 4(a).

[14] One cannot say definitively either way, as it is always at the C compiler's discretion.

FORCE_RET() is, then, a hack to overcome this problem, an attempt to force all paths through the function to a singular ending. The desired effect on the control flow graph is shown in Figure 3(b). GCC's code generator, then, when presented with such a function, apparently always places an end at the highest address of the function (though this is by no means strictly necessary). Specifically, FORCE_RET() is defined (at dyngen-exec.h:197) to be

```
__asm__ __volatile__("" : : : "memory");
```

This is an empty inline asm block decorated (with __volatile__) so that block motion and lifting is disabled in GCC. FORCE_RET()s are placed only where necessary, presumably determined by intuition or debugging. However, when it works, it works: op_arpl with FORCE_RET() compiles as in Figure 4(b), having only one return instruction at its highest address.

3.3 Register Allocation

For performance, QEMU typically binds the temporary variables and (some subset of the registers) of the guest machine into the host's registers while running the translated code.

This is achieved in GNU C by declaring variables with an extended syntax (in, for example, target-i386/exec.h:46):

```
register target_ulong T0 asm(AREG1);
```

where AREG1 is determined by dyngen-exec.h in a host-specific way. On X86-32, for example, AREG1 is defined to be ebx. GNU C semantics are that globals of this form reserve the register for the entire program.

Since various micro-ops either are stubs around calls to helper functions or may call helper functions or call out to raise exceptions in certain paths, QEMU choses only callee-save registers according to typical calling convention[15].

The simplest, if slowest, mechanism for solving this particular problem is to avoid the explicit use of registers altogether. There is extant code in QEMU (see, for example, target-i386/exec.h:38-40) for the case where the guest registers are larger than the host's, and so the temporaries Tn must be held in memory. The only change that would be necessary to make this case be totally free of explicit register assignments would be to move the CPU's state pointer back to global store.[16] Some limited testing of a QEMU built on a Linux host with a global environment variable seems to indicate that this does, indeed, work.

3.4 Relocation

QEMU makes use of the host platform's ability to carry out dynamic loading (or separate compilation) to allow its micro-ops to make function calls and reference global variables. Further, the mechanism is used "abusively" to load constants and for the implementation of non-local control flow.

3.4.1 Relocation of Functions and Globals

On many architectures, including X86, the default addressing mode for subroutine calls is relative to the instruction pointer. Since the dynamic translator is copying code, the instruction pointer will be different than the compiler anticipated, and the offset must be corrected in order for the code to work. Concretely, the op_cpuid function in QEMU's binary (compiled with GCC 4.1.3 on X86-64) looks like

```
000000000049dbc8 <op_cpuid>:
  49dbc8:       48 83 ec 08             sub    $0x8,%rsp
  49dbcc:       e8 8f bc 00 00          callq  4a9860 <helper_cpuid>
  49dbd1:       48 83 c4 08             add    $0x8,%rsp
  49dbd5:       c3                      retq
```

Notice that the machine representation of the callq is actually "the address of the end of this opcode (0x49dbd1) plus 0x0000bc8f" which gives 0x4a9860, or helper_cpuid. Thus the translator needs not only the compiled output from each micro-op C function but also the information about which parts of the binary must be rewritten in which way. This is exactly the relocation meta-data. To ensure that the compiler generates relocation records, helpers are defined in a separate C file from the micro ops.

Explicitly, the generated C part of the dynamic translator for emitting a op_cpuid micro-op is

```
extern void op_cpuid();
extern char helper_cpuid;
memcpy(gen_code_ptr, (void *)((char *)&op_cpuid+0), 13);
*(uint32_t *)(gen_code_ptr + 5) = (long)(&helper_cpuid) - (long)(gen_code_ptr + 5) + -4;
gen_code_ptr += 13;
```

Here, five bytes into the host instruction stream, the dynamic translator will land a computed expression such that at runtime the call is correctly dispatched to helper_cpuid.

[15] This has not been thoroughly verified, but it is the case at least on X86-32, X86-64, and PPC.

[16] At the moment QEMU does not support multiple processors in the guest, so while this would move the state pointer from per-CPU to per-process storage, it is hoped that this move would not alter semantics or correctness of the program.

3.4.2 Relocation to Simulate Immediate Parameters

The X86 micro-op corresponding to an "immediate load long" is `op_movl_T0_imu` (`target-i386/op.c:427`), which, with some explanatory definitions (from `dyngen-exec.h`) above is:

```
static int __op_param1;
#define PARAM1 ((long)(&__op_param1))
void OPPROTO op_movl_T0_imu(void)
{
    T0 = (uint32_t)PARAM1;
}
```

The code, as written, simply loads the address of a global variable, `__op_param1` into the temporary `T0`. However, this is not quite its use. Since this global is subject to relocation and link time, dyngen has a handle into the translation and can control exactly the value that is loaded to the register. Explicitly, this compiles (again, with GCC 4.1.3 on X86-64) to

```
0000000000497019 <op_movl_T0_imu>:
  497019:   44 8d 3d 7c 3e 27 02  lea  36126332(%rip),%r15d  # 270ae9c <__op_param1>
  497020:   c3                    retq
```

Here again we see indirection relative to the instruction pointer – "to load the value 0x270ae9c, add 0x02273e7c to the current instruction pointer, 0x497019" – though one could imagine instead that the compiler and linker may have emitted an absolute load. Either case would suffice, as the relocation data allows the dynamic translator to place any value into `T0`. The C code generated to emit `op_movl_T0_imu` is

```
long param1;
extern void op_movl_T0_imu();
memcpy(gen_code_ptr, (void *)((char *)&op_movl_T0_imu+0), 7);
param1 = *opparam_ptr++;
*(uint32_t *)(gen_code_ptr + 3) = param1 - (long)(gen_code_ptr + 3) + -4;
gen_code_ptr += 7;
```

We see that three bytes into the host opcode stream a computed value will be landed such that at execution time the desired value of the parameter (a scalar value, such as `$0x5` or `$0x2BADD00D`, not the address of any particular symbol) will arrive in `%r15d`, the host register assigned to back the micro-op virtual register `T0`.

The same mechanism is used to fold in addresses for non-local control flow. A slight variant is used to extract offsets into the translation buffers for switching off translation buffer chaining.

3.4.3 Relocation and Intermediate Formats of Compilation

Expanding on earlier discussion, dyngen currently takes the `.o` version of the micro-ops and emits C code to copy and do the relocation patching at runtime (see Figure 2). However, dyngen depends upon the intermediate format having both the native opcodes and the relocation data.

3.5 Translation Block Structure

Currently, translation buffers are pasted together centers[17] of each of the selected micro-op routines.

3.6 Micro-Op Non-local Control Flow

Despite that micro-ops are ostensibly written in C, use is made of GNU extensions to achieve non-local control flow transfer. This is used for two ends: exiting the translation unit, and branching inside one and to another translation unit.

3.6.1 Exiting The Translation Buffer

The first, exiting the translation unit, is comparatively simple, so we describe it first. The translator can emit code to bail from a translation unit at any point inside the unit. The micro-op for this makes use of a macro, `EXIT_TB()`, which is defined per-host-architecture to be a RET via inline assembler.

In order for this mechanism to work, it must be the case that the stack does not accumulate junk: when it comes time to return, the return address must be at the top of the stack. This unstated dependency happens to be satisfied by GCC's particular choice of compilation strategies for sufficiently simple functions like those of the micro-ops library.

[17] Complete with cream filling...

3.6.2 Jumping Within A Translation Buffer

Since the size of each micro-op core is known even before code generation has taken place, the code responsible for selecting micro-ops can keep track of the current offset into the translation buffer. This offset is captured whenever a label is desired; the list of labels is then passed to dyngen's emitted code and used to rewrite the instruction stream, thanks again to relocation records. Specifically, whenever a micro-op wishes to make a non-local jump, it uses the macro `GOTO_LABEL_PARAM(N)`, which is simply an inline assembler jump (defined per host architecture) to another abusive symbol, `__op_gen_labelN`. In response to this symbol, dyngen's emitted code pulls the Nth label from the given array and patches that in for the `JMP`'s target.

3.6.3 Chaining Translation Buffers

Non-local control transfer is further used across translation units to chain translations together to avoid having to return to the emulation loop. Such chains are undone on an interrupt to return control to QEMU. Each translation unit meta-data object has two patch locations for such chaining, providing up to two successors, as used by conditional jumps.

There are three implementations of QEMU's mechanism for translation buffer chaining, `GOTO_TB()`:

- A X86-specific version.
- A PPC-specific version.
- A GNU C implementation making use of GNU C's Labels as Values extension [2, Section 5.3].

While none of these implementations are suitable for use on Plan 9, it is instructive to consider at least one for concreteness. Tragically, all of the mechanisms here are full of horrors: the host-specific versions "know" which type of relocation records will be emitted by the linker in response to their code, and the GNU C implementation is remarkably odd.

The X86-specific version is inline assembler but probably simpler to explain than the GNUisms in the GNU C implementation. The inline assembler (from `exec-all.h:333`), with some manual preprocessing, is:

```
.section .data
__op_label##n#.op_goto_tb##n :
  .long 1f
.section .text
  jmp __op_jmp##n
1:
```

Here `n` is a parameter to `GOTO_TB()`; it is either 0 or 1 depending on which meta-data slot is being used for this jump[18]. What this achieves is to place a jump instruction with a destination determined by relocation, using another class of abusive symbols, `__op_jmpN`. Dyngen places the address of the patch location into an array for QEMU's use. The symbol placed in .data is yet another abusive class, `__op_labelN` to which dyngen responds by producing code to *export* the address of the relocation (the address of the instruction after the `jmp`) for the emulator's use.

After code generation, the translation block's chains are reset, meaning that for the host-specific versions, the jump location is patched with the address of the next instruction. After the micro-op containing `GOTO_TB()`, the translator will have placed micro-ops to store the instruction pointer and return from the translation unit back to the control loop. Upon either finding the next translation in cache or translating it anew, the jump location will be patched to be the head of that unit and the translation units' meta-data will be updated to show that they are linked. Thus, the next time this translation unit runs, it will jump directly to its successor, rather than have to involve the main loop.

4 Achieving Dynamic Translation on Plan 9 Hosts

4.1 Compiling The Dynamic Translator

FIXME

4.2 Requirements of Micro-Op Control-flow Graphs

4.2.1 kencc and Serializability

We have observed that `cc` – in particular the loader – apparently tends not to produce serializable routines, even from code with "obviously serializable" control graphs such as those in Figure 3(b). Further, it is observationally indifferent to syntactic "suggestions." This seems to stem from Plan 9's relatively unique calling convention, whereby most functions do not need prologues or epilogues. For example, with and without the label and `goto`s, the code of

[18] This example is marginally simplified; the micro-op name, here `op_goto_tbN`, is also a parameter to the macro. For the moment, though, there are no other consumers and so this example suffices.

Figure 5(a) compiles by `8c` into the intermediate representation shown in Figure 5(b) but is loaded by `8l` into the host code shown in Figure 5(c).

Removing the statement `global2 = 1;` merely removes its corresponding instruction from the emitted code but does not change the result's structure. Thus defining `FORCE_RET()` to be a statement with side-effects is insufficient under `cc`. Such productions are not suitable for dyngen's use as they always end in the middle.

For `8l`, the relevant code motion is carried out in `pass.c:/^xfol`. Some investigatory effort towards modifying this routine to produce serializiable functions, but no meaningful results have been achieved. However, it has not been deemed impossible either, so this avenue of attack remains open. Also remaining is to investigate other loaders.

```
int global, global2;
void quux(int a)
{
  if (a > 0) {
    global = 0;
    goto out;
  } else {
    global = 1;
    goto out;
  }
out:
  global2 = 1;
  return;
}
```

(a) Example micro-op like code.

```
TEXT quux+0(SB),0,$0
CMPL a+0(FP),$0
JLE  ,4(PC)
MOVL $0,global+0(SB)
JMP  ,3(PC)
JMP  ,3(PC)
MOVL $1,global+0(SB)
JMP  ,1(PC)
JMP  ,1(PC)
MOVL $2,global2+0(SB)
RET  ,
RET  ,
```

(b) Intermediate output of `8c`.

```
quux      CMPL a+0x0(FP), $0x0
quux+0x5  JLE  quux+0x1c(SB)
quux+0x7  MOVL $0x0, global(SB)
quux+0x11 MOVL $0x2, global2(SB)
quux+0x1b RET
quux+0x1c MOVL $0x1, global(SB)
quux+0x26 JMP  quux+0x11(SB)
```

(c) Final output produced by `8l`.

Figure 5: Demonstrating `cc`'s charmingly unique output

4.2.2 Alternatives

Since the micro-ops are all built at once (per guest architecture), it is possible that we could add a loader flag to ensure that all functions had only one return, placed at their highest address. This would allow us to define away `FORCE_RET` and trust the loader to do the right thing, rather than scatter `FORCE_RET`s wherever necessary whenever the compiler or loader changed behaviors. However, since we cannot load an already loaded program, an additional program would have to extract the fully loaded, modulo relocation, micro-op bodies and generate C files containing the host code as data to be compiled into QEMU.

It may also be possible to shim an intermediate program between the compiler and the loader, rewriting the intermediate format so that the loader produces serializable routines. This would be akin to the syntactic "suggestions" attempted with the `goto`s in 5(a), but at the assembler level. From investigation of `8l` this seems to be more difficult than the loader flag above.

Additional discussion can be found in Section 3.5.

Solution

See the discussion in Section 3.5.

4.3 Register Allocation

An alternative, if registerization is indeed desired,[19] could be crafted from the `extern register` variable class offered by `cc`. However, this class works correctly only when the entire program is compiled with all such declarations available for all compilation units. If at build time, we build all of QEMU's dependencies, such as `libc` and `libdraw`, with a modified `u.h` that includes the `extern register` declarations, this should suffice. This may make debugging the resulting executable more painful as the acid definitions will differ from the ones of the system library.

Solution

- The simplest solution may well be to avoid explicit register allocation altogether. There is extant code in the QEMU code base to do this.

[19] Rules of optimization: don't do it, and, for experts only, don't do it *yet*.

- Since registerization is likely to provide some non-trivial speedup of guest code, we may avail ourselves of the `cc`'s `extern register` storage class. However, the easiest way to meet the requirement of universal exposure to these declarations will be to build our own `libc`, `libdraw`, and other libraries we build.

4.4 Relocation

4.4.1 Relocation and Intermediate Formats of Compilation

`cc`'s intermediate format (the rough correspondence of a `.o` file) can still be relocated, as references are still by name, but does not contain native instructions, as those are only selected in full by the loader. Conversely, the loader generally fully specifies the layout of an executable and so discards the relocation data. However, it is hoped that the loader's understanding of dynamically loaded modules (from the delayed `dynld(2)` project[20]) will be sufficient to emit the relocation data that dyngen needs.

It seems that some small extensions to `dynld(2)` may be necessary. For the prurposes of constant loading, dyngen needs to know which symbols a relocation record references. This information is as readily available as anything is in the other executable formats dyngen understands, but `dynld(2)` currently does not offer any real semantic interpretation of relocation records to its callers. We hope that a function similar to `dynreloc()` (and taking the same parameters) which returned the relevant entry, if any, in the import table would suffice. Sadly, such a function would definitionally be per-architecture, but would be very simple.

4.4.2 Plan 9's Dynamic Load Facility

FIXME

Solution

- `cc`'s relocation capabilities are probably sufficient for the task at hand. If not, the changes necessary are probably small and can be contributed.

4.5 Translation Block Structure

We have trouble coring micro-ops on on Plan 9, so it would make sense to see if we could leave the functions unaltered as a first pass. In particular, this implies that we will have to ensure that we can move from one micro-op to the *next* (the not-taken successor) by *returning*. One layout of a translation buffer that would do this is

```
CALL &op_1
CALL &op_2
CALL &op_3
RET
op_1
op_2
op_3
```

This will slow down the simulation a little, but may be passable as a proof of concept. We still relocate the function bodies out but leave them containing `RET` instructions. Then they will return back to our chain of `CALL` instructions and all will be well. However, it is not clear that this layout deals well with inter-micro-op branches: we can simulate falling from one to the next just fine, but doing anything out of order looks hard. Instead of keeping the entire future on the stack, we could use a structure like

```
push l2
op_1
l2: push l3
op_2
l3: push l4
op_3
...
```

which keeps only the immediate successor micro-op on the stack. From an outside perspective, all we have done is grow the size of each micro-op by as many bytes as we need for the push. Since these bytes are a prefix to the micro-op, generated labels will naturally point there. Out-of-order control flow is available in this design using a few possibilities:

- Overwrite the successor value on the stack before returning.
- Manually pop the successor (in assembler, for example) before jumping to the appropriate label (and push).
- A `longjmp()`-style call to the next label which resets the stack.

[20] Which seems to be present in Inferno

What remains is to discuss non-local control flow more fully and check that it can, indeed, work with this translation buffer structure.

Solution

- We will (ab)use the stack to store the left successor micro-op's address before entering a micro-op.

4.6 Micro-Op Non-local Control Flow

4.6.1 Revisiting `GOTO_TB()`

The GNU C version causes GCC to emit an indirect jump (*e.g.*, `JMP %EAX`). It loads the target address directly from the translation buffer meta-data. It too exports a `__op_labelN` symbol, but rather than being patched

4.6.2 A Return To C

All of this poses a large problem: C proper (*i.e.*, GNU extensions and inline assembler aside) lacks any non-local control flow transfer mechanism other than a function call. It may also be useful to note that full functions written in assembler are available (and reasonably portable) on Plan 9; into this latter category fall `setjmp` and `longjmp`.

It should be noted that the use of jumps out of C function bodies is remarkably complex as it imposes many requirements of the machine code at the jump site. In particular, the stack pointer must be back where it was at function entry so as to avoid leaving trash around[21] Further, all live variables must have been committed to backing store as there will be no future point to do so; fortunately, this is required of global state at function call sites.

The semantics of `GOTO_TB()` make it more complex than just the constant jumps used by `GOTO_LABEL_PARAM()`, as it must be able to handle both the chained and unchained situations, and it must be easy for external code to toggle which behavior is active. The simplest way to achieve this may be to define `GOTO_TB(whichSuccessor)` using a host state in the current translation buffer meta-data structure and a host conditional, as in:

```
void *next = env->curr_tb->successor[whichSuccessor];
if(next)
  magic_jump_to(next);
```

This mechanism is slower (incurring the cost of a conditional on all paths) but makes use of only "normal" relocation (for `env` and whatever function serves the roll of `magic_jump_to`).

4.6.3 Using `longjmp()` Everywhere

C proper and kencc do not allow us to jump to arbitrary pointers or land inline assembler[22]. This and lack of GNU extensions rule out all current mechanisms within QEMU for achieving non-local control flow. In light of all of the constraints above, it seems easiest to (ab)use `longjmp()` to achieve all our non-local control flow needs. It is a simple, well-documented mechanism which gives us explicit control over both the stack and instruction pointers.

We may make a small modification to the CPU execution loop and enable the use of `longjmp()` to return from a translation block. This will require storing a jump buffer in the host state associated with the current guest CPU. Explicitly, the emulator loop now calls `setjmp(&env->exitjbuf);` before calling a translation buffer, and `EXIT_TB()` becomes

```
longjmp(&env->exitjbuf);
```

Normal relocation will suffice to allow access to `env`.

We can FIXME

Both jumps to other translation units and jumps to micro-ops within this translation unit can make use of `longjmp()`'s ability to reset the stack to alleviate concern over micro-ops use of stacks, as long as we first store it on entry into a translation buffer.

4.6.4 Alternatives

STOP READING

Solution

- We can offer an immediate solution to `EXIT_TB`, namely `longjmp()`.

[21] It is not strictly *required* that we not leave trash on the stack, but it is generally considered rude and would only serve to increase the cache footprint of the translation buffer.

[22] While the GCC developers doubtless view the absence of these extensions as bugs, one may achieve enlightenment if one views them as features.

5 Other Porting Issues

5.1 Register Calling Conventions

The software MMU code makes use of a GCC extension[23] to modify the register usage of its calling convention for several load and store instructions. Further, the modified register convention is hard-coded in hand-written inline assembler for their callers. However, it seems that most of this can be switched off and the C version used instead.

Solution

This is an optimization used by X86-on-X86 simulation and may be considered premature optimization for the purposes of the initial port.

5.2 Translation Block Program Counter

Helper code for the translated micro-op stream frequently wishes to know the actual program location, and so uses GCC's `__builtin_return_address(0)` function to extract it from the stack.

Solution

It should be straightforward to replace this with `getcallerpc`.

5.3 Explicit Branch Prediction Overrides

Some use is made of GCC's `__builtin_expect(v,c)` extension to provide hints to the processor's branch predictor. This may be dealt with by `#define`-ing away the annotation or adding branch prediction hints to `cc`. The latter sounds like a project for another time, if ever a convincing case for their use is made.[24]

Solution

This may also be viewed as premature optimization for the purposes of the initial port and so removing the annotation should suffice.

5.4 Memory Management

QEMU supports both a "softmmu" mode and a "user" mode emulation strategy. The former emulates a full memory management unit (with translation cache), while the latter uses `mmap` and `mprotect` to host a system inside user usable address space. This is intended for running executables compiled for one architecture on another, under the same operating system.

The absence of `mmap` could be overcome by use of `segattach`, but no mechanism parallel to `mprotect` exists on Plan 9. Fortunately, "user" mode emulation is not likely attractive to Plan 9 users and so may be considered unnecessary to port[25].

Solution

It seems that system emulation mode uses no advanced memory tricks and so nothing beyond `libc`'s standard allocator functions will be necessary.

5.5 Locking

QEMU uses some limited test-and-set locking techniques for threading support in "user" emulation mode (not yet in "system" mode; SMP is implemented by round-robin emulation of the CPUs) and for CPU interrupt management. Currently every architecture codes in inline asm the appropriate test-and-set mechanism for implementing locks.

Some locking is sprinkled around the code in what seems to be active development towards taking advantage of multiple host processors. However, this code is incomplete which may pose problems; see Section 5.6.

[23] The rather ugly `__attribute((regparm(N)))` which specifies that the first `N` parameters should be passed as registers.

[24] There have been discussions on GCC's mailing list about using branch predictor hints for pointers that result from `malloc()`, which strikes this author as remarkably silly. Hints also appear as decoration in Linux but this author is not aware of performance figures demonstrating a non-decorative utility.

[25] While it is acknowledged that "user" mode is insecure – the guest code can modify QEMU's host code – it is still actively developed upstream. The previous assertion that it was deprecated therefore seems misleading.

Challenge	Current Favored Solution
Control Flow	A modified translation block structure.
Register Allocation	Optimization; defer.
Register Calling Convention	Optimization; defer.
Relocation	The extant `dynld(2)` mechanisms provide sufficient relocation meta-data.
Program Counter	`getcallerpc` will suffice.
Branch Prediction Overrides	Optimization; defer.
Memory Management	It is believed that `libc`'s standard allocator will suffice.
Locking	Trivially reimplemented using `libc`'s `_tas`.
Signals	Reimplemented in terms of notes.
Asynchronous I/O	Optimization; defer.

Table 1: Summary of identified difficulties and the proposed mechanism of solution.

Solution

Plan 9's libc provides a `_tas()` function which implements test-and-set.

5.6 Interacting With The Outside World

QEMU makes use of signals and POSIX AIO on UNIX and UNIX-like hosts[26] to deliver interrupts. Notable consumers include the QEMU timer/clock driver (which uses SIGALARM) and the block device driver (which prefers to use POSIX AIO). Interestingly, it seems that file descriptors from streams (e.g., the UI connection to X or a VNC client, emulated network sockets, emulated serial ports, etc.) are polled via select only after a timer tick.

Every code path which wishes to deliver an interrupt to the guest CPU must call `cpu_interrupt()`. There is a comment (`vl.c:7176`) in some Windows specific code which reads

```
/* Note: cpu_interrupt() is currently not SMP safe, so we force
   QEMU to run on a single CPU */
```

This is quite the understatement: currently there is no synchronization between the cpu emulator loop's and the interrupt delivery path's attempts to modify translation buffer chaining. This mostly works as currently QEMU in "system" mode is single-threaded, implicitly serializing everything including signal delivery. However, it is not clear that the current code is immune to interrupt deferral for arbitrary amounts of time or loss[27].

Plan 9's notes also act as interrupts rather than acting in separate threads (as in Windows), so a straightforward transform should yield code that is as correct on Plan 9 as it is on other platforms.

Solution

The Plan 9 note mechanism should suffice for timer management. The block device code appears to have some way of avoiding use of AIO, but details are fuzzy.

5.7 User Interface

Tragically, little thought has been given to this. However, since Uriel has an SDL port to Plan 9, it is sincerely hoped that little effort is necessary to get at least a simulated VGA display, keyboard, and mouse available.

It is further hoped that with relatively little effort QEMU can be taught about `/net` for user network emulation.

6 Summary

This paper presents an initial attempt at a strategy map for porting QEMU to Plan 9. From reading the QEMU paper [1], reading of QEMU code, some reading of the compiled binaries, and some hints as to where to begin, a series of potential issues were identified. For each, at least one solution is herein proposed for review; whenever possible, an effort has been made to identify other possible solutions as well. It should be noted that this is by no means an exhaustive list. For quick reference, the favored solution for each identified problem is tabulated in Table 1.

[26] And similarly complex mechanisms on Windows hosts
[27] See Appendix A

References

[1] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. *USENIX*, 2005. URL `http://www.usenix.org/publications/library/proceedings/usenix05/tech/freenix/full_papers/bellard/bellard.pdf`.

[2] *GCC 4.2 Manual*. Free Software Foundation, Inc. URL `http://gcc.gnu.org/onlinedocs/gcc-4.2.0/gcc/`.

[3] Kevin Lawton. *BOCHS, The Open-Source IA-32 Emulation Project*. URL `http://bochs.sourceforge.net/`.

[4] Johannes Schindelin. Porting QEMU to new CPU. 2004. URL `http://libvncserver.sourceforge.net/qemu/qemu-porting.html`.

A QEMU Interrupt Bug

The CPU interrupt dispatch mechanism's goal is to unchain whatever translation block is running and force control flow to return to the main CPU execution loop. Its basic structure is:

1. Mark an interrupt-specific flag.
2. Fetch the current translation block.
3. Remove the environment's pointer to the current translation block.
4. Recursively unchain the current translation block.

The main CPU execution loop, to which we are trying to ensure control flow returns, has the basic structure:

1. Check for interrupts in the flag word and if any are set, handle them.
2. Find or generate the next translation block.
3. If we are chaining (*i.e.* not on an exception path and the just-executed translation block left behind patching instructions), patch the current translation block with a chain to the next translation block.
4. Set the next translation block as the current.
5. Run the current translation block.

Note that there is a clear point, just after step 3 of the CPU execution loop, which we may call the key point, where all of the following conditions can hold:

1. A commitment has been made to which translation block will run next, but it is not yet considered the current translation block.
2. It might be the case that the current translation block is not chained to the new translation block. (More strictly, it might be the case that the current translation block's transitive closure under chaining does not include the new translation block.)
3. The next translation block may have extant chaining patches, if it was pulled from cache. In the worst case, it may be part of a cyclic chain.

The first condition ensures that we have already selected our next translation block and are not able to select a different one. The second condition implies that the interrupt dispatch mechanism's recursive unchaining may not affect the already-selected translation block's chaining. The addition of the third condition yields that there are cases where interrupt dispatch is deferred until the next asynchronous interrupt (*e.g.*, timer tick). If interrupts are not queued, but merely use the flag bits to signal their pending status, then this bug may additionally imply loss of interrupts.

The fix seems to be either disabling asynchronous signals during parts of the CPU execution loop or to re-check for interrupts after the next translation block has been set as the current. The former is slow, imposing two system calls per pass through the CPU execution loop. The latter implies more code re-structuring than I wish to do, as the bigger problem of the port remains.