Porting QEMU to Plan 9: Strategy

Nathaniel Wesley Filardo

July 10, 2007

1 Overview

QEMU is a fast, portable emulator of many architectures, including IA-32 (X86), PPC, and SPARC. It seems worth-while 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 Comparison to Bochs

Bochs [2] 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 used to produce a block of native code. These micro-ops range in complexity from simple simulated register transfer moves to integer and floating point math to memory load and store operations (which require simulating the guest architecture's paging mechanism). Translation is done per basic block of guest code and translations are cached for reuse. Between each translation unit simulator code runs to check for pending interrupts and to compute the next basic block.

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 the instruction and 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} temporary 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.
- Move the second operand into the 1^{st} temporary register.
- Do the core of the `ARPL` instruction. This micro-op demands that its operands are in temporary registers 0 and 1.
- Move the result from the 0^{th} temporary back to the original source. Again, this will be specialized *either* to a store or a register move.

These micro-ops will be translated to host machine code and stored in the translation cache for subsequent use.

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 includes a reasonable selection of devices likely to be supported by guest operating systems. Both Bochs and QEMU use BIOSes run inside the simulation to initialize certain parts of the hardware, a design decision which allows the device emulators to remain faithful to the original hardware.

[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.

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 clearly factored out into their own files and directories. All guests speak the same interface to the core, drivers, and dynamic translator. Of the some 111,000 lines of code[3] 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[4].

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 and some UNIXisms in the host driver code.

1.4 Roadmap

The remainder of this paper is broken down into sections detailing individual features of the codebase that would make compiling under Plan 9 and kencc difficult. Previously, Christoph Lohmann worked on porting the QEMU infrastructure code; Uriel has an initial port of SDL to Plan 9, which should help in porting the GUI. The majority of the paper, Section 2, is concerned with the dynamic translator, which has not been previously addressed and is quite thorny. Some additional discussion is devoted to memory management (Section 3) and locking (Section 4) behaviors of QEMU.

2 Dynamic Translator

2.1 Overview

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 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.

The compilation phases are shown in Figure 1. 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. For further discussion, see Section 2.2.4.

The emulation strategy during runtime is, approximately, a loop containing the steps

1. Check for pending guest interrupts. If any are present, store the current machine state and simulate the guest hardware's interrupt dispatch mechanism to determine the next block of code to run.
2. Consult the translation cache to see if we've done this before. If a cache is found, jump to step 6, otherwise continue.
3. While the current instruction does not alter control flow,
   (a) disassemble instructions of the guest machine
   (b) For each guest instruction, select the corresponding sequence of micro-ops, as in Section 1.1.
   (c) Copy the code for each of these micro-ops from the compiled version into the translation buffer.

[3] grep -c \;
[4] Of interest in particular to the Plan 9 community are its X86, ALPHA, MIPS, PPC, and possibly SPARC.

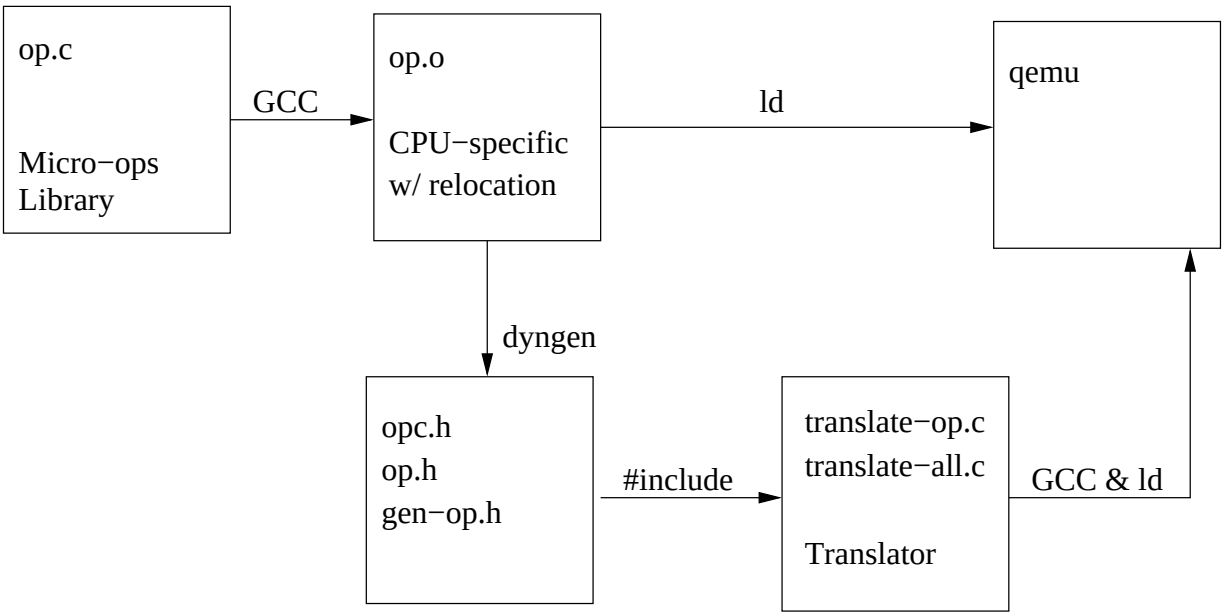


Figure 1: Compiling the micro-ops library.

4. Emit a host-specific "return" instruction at the end of the translation buffer, to return flow control to the translation engine.

5. Record meta-data about the block chaining along side the translation unit.

6. Jump into the translation unit.

2.2 Challenges

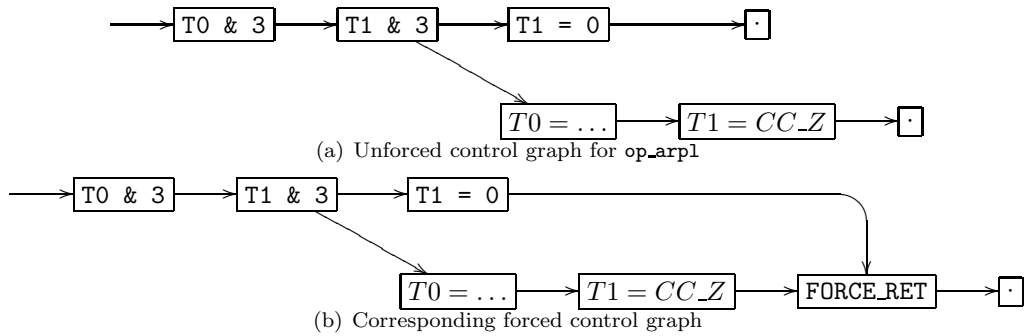
2.2.1 Micro-Op Control-flow Graph Requirements

The micro-ops are written in 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. 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.

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 2(a). In



(a) Unforced control graph for `op_arpl`

(b) Corresponding forced control graph

Figure 2: 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 3: GCC 4.1.3 on X86-64 compilation results for `op_arpl`.

this case, there may well be a "return" instruction in the middle of the host instruction stream, rendering the micro-op unsuitable for concatenation[5], as in Figure 3(a).

`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 2(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 3(b), having only one return instruction at its highest address.

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 2(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 `gotos`, the code of Figure 4(a) compiles by `8c` into the intermediate representation shown in Figure 4(b) but is loaded by `8l` into the host code shown in Figure 4(c).

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

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:/ẍfol`. 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 4: Demonstrating `cc`'s charmingly unique output

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 4(a), but at the assembler level. From investigation of `8l` this seems to be more difficult than the loader flag above.

Another approach may be to alter the specific structure of translation units. Currently, translation buffers are pasted together centers[6] of each of the micro-op routines. We could dodge all of this selective copying if, instead, our translation buffers were of the form

```
CALL op_...
CALL op_...
CALL op_...
RET
```

This will slow down the simulation a little, but may be passable as a first shot. Not all micro-ops are amenable to being called; this strategy also has further interactions with relocation; see the discussion in Section 2.2.4. It may be possible to restrict the set of functions that are `CALL`ed to those which do not have a unique return, potentially forming a neat way to side-step the problem.

Using the above `CALL`-chain, the stack grows a return address to the next call instruction for every dispatch. What if instead, we arranged for the stack to look like

```
&op_...
&op_...
&op_...
```

[6]Complete with cream filling...

before transfering control to the first micro-op in a translation buffer? Then that op's return would hand control immediately to the second, etc. We can arrange for this kind of stack layout by simply `PUSH`ing constants; in fact, we could consider each `PUSH` to be a micro-op in its own right. Under this scheme, we are even free to copy and relocate all we need, making a full translation buffer look like

```
...
PUSH &op_3
PUSH &op_2
op_1 body
op_2 body
...
```

We can then simply `CALL` into the head of such a translation buffer, which will arm the stack for us and then execute each micro-op by `RET`ing, and then return control flow to the point of the call. Since all micro-ops are void functions returning void, they will return the stack to the same state as on entry, so we need not fear frame shift. This scheme works essentially orthogonal to relocation and requires no modification to the loaders; its costs are slightly larger memory footprint of translation buffers, potentially slower translations[7], and the additional effort to create the `PUSH` chain.

Solution

- The initial implementation will go with the `PUSH`-chain variant outlined above. Subsequent work could follow any of the optimizations discussed above, those used by QEMU, or others as desired.

2.2.2 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[8].

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.[9]

An alternative, if registerization is indeed desired,[10] 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.
- 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.

[7] Though moden hardware make make this neglidgable.

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

[9] 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.

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

2.2.3 Register Calling Conventions

The software MMU code makes use of a GCC extension[11] 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.

2.2.4 Relocation

Some micro-ops make function calls to helpers and reference global variables. Thus, their compiled forms need to be rewritten using relocation information. 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.

Relocation of Functions and Globals

The X86 instruction `CPUID` is remarkably complex[12] and requires about 75 lines of C even in QEMU's simplistic implementation[13]. 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`).

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`.

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

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

[13] 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.

Relocation to Simulate Immediate Parameters

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.

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`.

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 1). However, dyngen depends upon the intermediate format having both the native opcodes and the relocation data.

`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[14]) 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.

[14] Which seems to be present in Inferno

Interaction With "Series of Calls" Style Translation Buffers

The decision to use a "Series of Calls" style translation buffer would alleviate some of the need for relocation. This has not been fully considered, but some thoughts are presented below.

- Immediate loads pose a problem regardless. Either one of the other mechanisms mentioned in passing (or perhaps another) would need to be constructed and implemented, or function bodies for immediate loads would still need to be copied and rewritten. The latter strategy turns the "Series of Calls" style into a "Series of Calls And Loads":

```
MOVL $0x2BADD00D, %r15d
CALL op_...
CALL op_...
MOVL $0x5, %r15d
CALL op_...
RET
```

- Assuming no immediate load micro-ops load into guest registers that are not host registers, the latter strategy would not mandate the reintroduction of global relocation.
- If micro-ops containing function calls are never copied (that is, we always call them), then function call relocation can go away. However, as function call and global relocation uses the same mechanism, this does not seem a sufficient motivation to choose "Series of Calls" style translation buffers.
- A desire to simulate SMP guests would entail either creating a different micro-op library for each guest CPU or enabling relocation of globals, regardless of translation buffer style.

The upshot seems to be that even with a modified buffer translation strategy, use of relocation data is still necessary, and so aiming to eliminate it may not be worth while.

Alternative Relocation Meta-Data Recovery

In absence of a `dynld` solution, it is possible that a more manual relocation mechanism could be developed. It should be easy to determine the set of symbols under consideration, or sufficiently straightforward to compile a list by hand, or some combination of these approaches. Through the use of some combination of `acid`, `mach`, `debugger`, `symbol`, and guesswork it may be possible to recover (to sufficiently high fidelity), from the compiled and loaded executable, the relocation meta-data. This could then be patched in to arrays in the executable file.

Note that this solution is hackish and clumsy, especially since the loader already has, by necessity, the information desired. Moreover, it represents a radical departure from the current workings of dyngen. Neither of these problems are necessarily show-stoppers, but they suggest that other approaches be tried first.

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.
- It might be possible to recover the necessary amount of relocation meta-data through force.

2.2.5 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`.

2.2.6 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.[15]

Solution

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

3 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 deprecated upstream for security concerns and so may be considered unnecessary to port.

Solution

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

4 Locking

QEMU uses some limited test-and-set locking techniques for threading support in "user" emulation mode (not yet in "system" mode, though SMP is on the radar) and for CPU interrupt management. Currently every architecture codes in inline asm the appropriate test-and-set mechanism for implementing locks.

Solution

We are not aiming to recreate the "user" emulation mode, and so locking may well prove unnecessary. When and if SMP guest support becomes available, the QEMU Plan 9 support code can be readily extended to use `QLock`s.

5 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.

5.1 Next Steps

1. Contact Charles Forsyth for details of `dynld` and feasibility of single-return loader flag.
2. Contact Uriel about SDL port for GUI, required for a later stage.

[15] 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.

Challenge	Current Favored Solution
Control Flow	Modifying the loader to produce code more similar to the form dyngen expects.
Register Allocation	Optimization; defer.
Register Calling Convention	Optimization; defer.
Relocation	It is hoped that the extant `dynld(2)` mechanisms in the loader 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	Unnecessary for the emulation modes being considered.

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

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

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