Executor Internals: How to Efficiently Run Mac Programs on PCs

Mathew J. Hostetter <mat@ardi.com>
Clifford T. Matthews <ctm@ardi.com>

After MacHack '96, this paper will be available from http://www.ardi.com

Executor is a commercial Macintosh emulator that uses no software from Apple, but is still able to run much 680x0 based Macintosh software faster on Pentiums than the same software runs on 680x0 based Macs. This paper contains some implementation details, including descriptions of Executor's synthetic CPU, graphics subsystem and debugging environment. Portability issues, current limitations and future plans are also presented.

Executor Overview

What Executor is

Executor is a commercial emulator that allows PCs to run many Macintosh applications. Executor does not require Macintosh ROMs or a Macintosh System file and contains no Appple code itself. Executor was written entiredly by engineers without Macintosh backgrounds who have not disassembled any of Apple's ROMs or System file.

Limitations

Because Executor was written strictly from publicly available documentation (Inside Macintosh, Tech. Notes, etc.), programs which make use of undocumented features of MacOS may fail under Executor. Furthermore, there are some portions of MacOS that we haven't implemented yet. Executor is sufficiently large that there are probably bugs in some of our code as well. We realize these are major limitations, but this paper is primarily concerned with implementation details that are interesting to our fellow programmers as opposed to feature sets and limitations which are of more concern to end users and our marketing department.

Design Goals

Our goal is for Executor to be accurate, fast and portable. Beyond that, completeness is a secondary issue.

Accuracy means that each subsystem that we implement should behave exactly according to the functional specs for the subsystem that we've derived from a combination of reading documentation, writing test cases and running programs under Executor.

Fast is harder to qualify. As programmers we like to use advanced techniques that will result in programs running under Executor as quickly as possible. Unfortunately, we have a limited number of engineer hours in a week and most engineering time is spent implementing new subsystems or finding and fixing subtle incompatibilities. We're proud of the speed that we've obtained so far, but we know that we can do better in the future.

Portability is the ability to support multiple platforms from the same source base. A platform is a combination of CPU, operating system and graphics device or windowing system. Executor currently supports Intel 80[3456]86 and compatible CPUs, Motorola m680[34]0 CPUs, the operating systems DOS, Linux and NEXTSTEP and can interact with VGA, SVGA, Display PostScript and X-Windows. To get the best performance on some architectures we do use architecture specific code, but we also write portable versions to be used where the platform specific versions can't be. Although not supported as a product, Executor was ported to DEC's Alpha, but since ARDI has no Alpha and DEC lost interest, the Alpha port is no longer current. Although not recently, ROMlib, ARDI's rewrite of the MacOS OS and Toolbox routines, has been ported to a wide variety of platforms, including MIPS , m88k , Clipper, IBM RT, SPARC and even VAX based systems.

Those three design goals have led us in the direction of dynamic code generation for both the 680x0 emulation and for our blitter. In both cases we use high level descriptions of what we want accomplished and then use special purpose tools at compile time to translate these high level

descriptions into constructs that we can then use at run time.

High level descriptions are less error prone, allowing us to document the semantics that we wish to see in our synthetic CPU or blitter using a special purpose language that is directly suited to the task at hand, rather than a general purpose language like C or the traditional language of speed freaks -- assembler.

High level descriptions also lend themselves to portability. We have our tools generate portable constructs for the general case and, with a little more programming effort, faster architecture specific constructs for the architectures that we consider most important.

Since the conversion from high level description to useful construct takes place at compile time, there is no need to worry about the CPU cycles spent doing the mapping. This allows us to design our code by thinking: "At *runtime*, what would be the optimal instruction sequence to perform a specific task?" Once we know the answer to that question we can ask: "How can we represent at a high level, the task is being accomplished by that optimal set of instructions?". Then, the final question is "Given what we want to generate and how we want to represent it, what does the compile time mapping look like?". The entire time we're pondering those three questions, we're keeping accuracy, portability and efficiency in mind.
Executor Subsystems
Synthetic CPU

Overview

Syn68k is the name of the synthetic CPU that Executor 2 uses. Syn68k is both highly portable and fast. The portable core of Syn68k, which works by dynamically compiling 680x0 code into an efficient interpreted form, was designed to run on all major CPU's. On supported architectures, Syn68k can also translate 680x0 code into native code that the host processor can run directly.

Syngen

Syngen analyzes a lisp-like file describing the bit patterns and semantics of the 680x0 instruction set and produces lookup tables and C code for the runtime system to use. The code and tables generated by syngen depend somewhat on the characteristics of the host processor; for example, on a little endian machine it is advantageous to byte swap some extracted 680x0 operands at translation time instead of at runtime.

The 680x0 description file can describe multiple ways to emulate any particular 680x0 opcode. The runtime system looks at what CC bits are live after the instruction and chooses the fastest variant it can legally use. In Figure 1, we have two CC variants of lsrw; one computes no CC bits, and the other computes all of them.

The 680x0 description file can also specify which 680x0 operands should be "expanded" to become implicitly known by the corresponding synthetic opcode. For example, fully expanding out "addl dx,dy" would result in 64 synthetic opcodes, one for each combination of data register operands. This results in smaller and faster synthetic opcodes at the expense of increasing the total number of synthetic opcodes. To conserve space, we only "expand out " common 680x0 opcodes. On host architectures where we can compile to native code, we don't waste space by "expanding out" common synthetic opcodes.

Interpreted Code

Our interpreted code consists of contiguous sequences of "synthetic opcodes" and their operands. Syngen can generate ANSI C, but when

```
(defopcode lsrw_ea
 (list 68000 amode_alterable_memory () (list "1110001011mmmmmm"))
 (list "-----" "-----" dont_expand
      (assign $1.muw (>> $1.muw 1)))
 (list "CN0XZ" "-----" dont_expand
      (list
       (assign ccx (assign ccc (& $1.muw 1)))
                           (ASSIGN_NNZ_WORD (assign $1.muw (>> $1.muw 1))))))
```

Figure 1. Syn68k description of lsrw

compiled with GCC it uses C language extensions that make synthetic opcodes be pointers to the C code responsible for interpreting that opcode. This "threaded interpreting" entirely eliminates switch dispatch and loop overhead.

Native Code

For the 80x86 architecture, Syn68k supports an optional architecture-specific native code extension that tries to generate native code whenever possible. In those rare cases when it cannot, it reverts to our interpreted code. Since Syn68k supports both native and synthetic code, the runtime system automatically inserts gateways between the two whenever there is a transition.

Three major problems make translating 680x0 code to 80x86 code difficult:

- The 80x86 has only 8 registers, while the 680x0 has 16.
- The 80x86 is little endian, while the 680x0 is big endian.
- The 80x86 does not have general-purpose postincrement and predecrement operators, which are used frequently in 680x0 code.

On the other hand, several factors make the job easier than it would be for RISC machines:

- The 80x86 has all of the CISC addressing modes commonly used in 680x0 code.
- The 80x86 has CC bits that map directly to their 680x0 counterparts (except for the 680x0's X bit).
- The 80x86 supports 8-, 16- and 32-bit operations, (although it can only support 8 bit operations on four of its registers).
- The 80x86 and 680x0 have analogous conditional branch instructions.
- The 80x86 allows unaligned memory accesses without substantial overhead.

The toughest problem is the lack of registers. On 32-register RISC architectures it's easy to allocate one RISC register for each 680x0 register, but on the 80x86 a different approach is needed. The obvious solution is to perform full-blown inter-block register allocation, but we fear that using traditional compiler techniques would be unacceptably slow.

For now, we have adopted a simple constraint: between basic blocks, all registers and live CC bits must reside in their canonical home in memory. Within a block, anything goes. So what liberties does Syn68k take within a block?

The 80x86 register set is treated as a cache for recently used 680x0 registers, and the 80x86 CC bits are used as a cache for the 680x0 CC bits. At any particular point within a block, each 680x0 register is either sitting in its memory home or is cached in an 80x86 register, and each live 680x0 CC bit is either cached in its 80x86 equivalent or stored in its memory home. Cached registers may be in canonical form, may be byte swapped, may have only their low two bytes swapped, or may be offset by a known constant from their actual value.

Each 680x0 instruction can require that 680x0 registers be cached in particular ways. For example, `movel d0, mem` requires d0 to be cached in big endian byte order. The compilation engine generates the minimal code needed to satisfy those constraints and then calls a sequence of routines to generate the native code. As each 680x0 instruction is processed, each 680x0 register's cache status is updated. Dirty registers are canonicalized and spilled back to memory at the end of each block (or when we run out of 80x86 registers and we need to make room).

We allow 680x0 registers to be cached with varying byte orders and offsets so that we can perform the optimizations of lazy byte swapping and lazy constant offsetting. If the 680x0 program loads a register from memory and then ends up writing it out later, we avoid unnecessary byte swaps by not canonicalizing the value immediately. Lazy constant offsetting mitigates

```
pea         0x1
pea         0x2
pea         0x3
pea         0x4
...
```

becomes this 80x86 code:

```
movl  _a7,%edi
movl  $0x01000000,-4(%edi)    ; "push" big-endian constant
movl  $0x02000000,-8(%edi)
movl  $0x03000000,-12(%edi)
movl  $0x04000000,-16(%edi)
... <more uses of a7 may follow, and they'll use %edi>
subl  $16,%edi
movl  $edi,_a7
...
```

Figure 2. Lazy Constant Offsetting

the overhead of postincrement and predecrement side effects. Figure 2 is an example of lazy constant offsetting.

As mentioned above, we use the 80x86 condition code bits as a cache for the real 680x0 CC bits. Although live cached CC bits are occasionally spilled back to memory because some 80x86 instruction is about to clobber them, this trick almost always works. Using 80x86 CC bits, we can frequently get away with extremely concise code sequences; for example, a 680x0 compare and conditional branch becomes an 80x86 compare and conditional branch.

Self-modifying Code

Like most dynamically compiling emulators, Syn68k doesn't detect self-modifying code; the overhead is too high. Fortunately, self-modifying programs don't work on the real 68040 either. We rely on the program making explicit system calls to flush the caches whenever 680x0 code may have been modified or created. Some programs (like HyperCard) flush the caches very often, which can cause real performance headaches if code is continuously recompiled. We have solved this problem by checksumming 680x0 blocks as they are compiled and only decompiling blocks which fail their checksums. This optimization alone sped up some HyperCard stacks by a factor of three or so.

Examples

Figure 3 contains two sample 680x0 code sequences from real applications, and the 80x86 code that Syn68k generates for them. We chose these code sequences specifically to showcase several of the techniques we use, so you shouldn't use them as a substitute for benchmarks. Not all 680x0 code translates as well as these examples do, but these examples are far from exotic.

Example 1 (Solarian):

680x0 code:

```
addqb #1,a4@(1)
movel #0,d0
moveb a4@,d0
swap  d0
clrw  d0
swap  d0
asll  #2,d0
lea   a5@(-13462),a0
addal d0,a0
moveal      a0@,a0
movel #0,d0
moveb a4@(1),d0
cmpw  a0@,d0
bcs   0x3fffee2
```

80x86 code:

```
movl  _a4,%edi           ; addqb #1,a4@(1)
addb  $0x1,0x1(%edi)
xorl  %ebx,%ebx          ; movel #0,d0
movb  (%edi),%bl         ; moveb a4@,d0
rorl  $0x10,%ebx         ; swap d0
xorw  %bx,%bx            ; clrw d0
rorl  $0x10,%ebx         ; swap d0
shll  $0x2,%ebx          ; asll #2,d0
movl  _a5,%esi           ; lea a5@(-13462),a0
leal  0xffffcb6a(%esi),%edx
addl  %ebx,%edx          ; addal d0,a0
movl  (%edx),%edx        ; moveal a0@,a0
xorl  %ebx,%ebx          ; movel #0,d0
movb  0x1(%edi),%bl      ; moveb a4@(1),d0
bswap %edx               ; cmpw a0@,d0
movw  (%edx),%cx
rorw  $0x8,%cx
cmpw  %cx,%bx
movl  %edx,_a0           ; <spill dirty 68k
movl  %ebx,_d0           ;  registers back to memory>
jb    0x6fae0c           ; bcs 0x3fffee2
jmp   0x6faf0c           ; <go to "fall through" code>
```

Example 2 (PageMaker):

680x0 code:

```
movel #0,d2
moveb d0,d2
lslw  #8,d0
orw   d0,d2
movel d2,d0
swap  d2
orl   d2,d0
movel a0,d2
lsrb  #1,d2
bcc   0x3fffed4
```

80x86 code:

```
xorl  %ebx,%ebx          ; movel #0,d2
movl  _d0,%edx           ; moveb d0,d2
movb  %dl,%bl
shlw  $0x8,%dx           ; lslw #8,d0
orw   %dx,%bx            ; orw d0,d2
movl  %ebx,%edx          ; movel d2,d0
rorl  $0x10,%ebx         ; swap d2
orl   %ebx,%edx          ; orl d2,d0
movl  _a0,%ecx           ; movel a0,d2
movl  %ecx,%ebx
shrb  %bl                ; lsrb #1,d2
movl  %ebx,_d2           ; <spill dirty 68k
movl  %edx,_d0           ;  registers back to memory>
jae   0x3b734c           ; bcc 0x3fffed4
jmp   0x43d48c           ; <go to "fall through" 68k code>
```

Figure 3. 680x0 -> 80x86 examples

Graphics

SVGA Graphics

The DOS world is one of standards. *Many* standards. Standards made by engineers who were even more short-sighted than the folks who brought you ROM85, only to be replaced by SysEnvirons which was then replaced by Gestalt. The first color graphics adapter for the PC (CGA) was replaced with EGA, which was then replaced by VGA, which eventually gave way to several different Super Video Graphics Array (SVGA) cards.

SVGA cards have a couple of properties that make them less than perfect targets for the output of Macintosh emulators. First, the default is for SVGA's video memory to only be mapped into the PC address space through a 64k window (or bank). If you want to display 640x480x8 bits you need to write 64k of information to the 64k screen address range, then tell the video card that you want that same address to represent a different 64k chunk of the screen, then you write to that address range again, then you switch banks again, and so forth.

The second major complication is that under DPMI, the address space that contains the SVGA video memory is not in the same address space that a 32-bit application uses. For those of you used to programming in a flat address space, it might be hard to believe that you need special machine language address space overriding prefixes to access screen memory, but under DPMI 0.9 (which is the version of DPMI that Microsoft supports; we wouldn't have to do this under 1.0) "selector" overrides really are necessary.

Blitter Overview

A Region is a data structure that describes a set of pixels. Regions can be created by the application by calling various MacOS toolbox routines. In addition the toolbox routines themselves sometimes create Regions for their own purposes.

A blitter is a set of software or hardware which takes sets of bits, representing pixels, and combines them with other sets of bits in a variety of different ways. A Region blitter is a blitter that processes pixels by Regions (rather than by rectangles or rectangle lists).

A Simple Blitter

One way to write a simple Region blitter is to start with a subroutine that parses the start/stop pairs of a Region scanline and draws the corresponding pixels. This subroutine is then called once for each row of pixels to be displayed.

Unfortunately, this approach is slow since each scanline gets re-parsed every time it is drawn. The Region for a 300 pixel tall rectangle consists of a single scanline with a repeat count of "300"; this "simple Region blitter" will parse that scanline 300 times! That's a lot of redundant work.

There are many possible ways to get away with parsing each scanline only once. One approach is to convert the start/stop pairs into a bit mask where the bits in the mask correspond to the bits in the target bitmap that are to be changed. The inner blitting loop then becomes an exercise in bitwise arithmetic. In C, such a loop might look something like this:

```
for (x = left; x < right; x++)
   dst[x] = (dst[x] & ~mask[x])
          | (pattern_value & mask[x]);
```

That's not bad, but we can do better.

A Dynamically Recompiling Blitter

Using an explicit bit mask array is unnecessarily slow in the common case of filling a rectangle. For a rectangular Region, mask[x] is usually all one bits, making the bit munging a waste of time. And even when the masks are never solid (e.g. when drawing a thin vertical line), this technique is still unnecessarily slow. As it turns out, even the cycles the CPU spends loading mask bits from memory are unnecessary. Furthermore, even if we were satisfied with the level of performance that C code like the above provides, we couldn't use it on a stock SVGA system because it wouldn't know how to access the SVGA portion of memory.

Executor's blitter uses the techniques of partial evaluation and dynamic code generation to eliminate redundant work and also give us access to SVGA memory. On the 80x86 each scanline is quickly translated into executable code, and that code gets executed once each time the scanline needs to be drawn. On non-80x86 platforms, each scanline is compiled into threaded code which is executed by a machine-generated interpreter to draw the scanlines.

Before describing how the dynamic compilation process works, let's take a look at an example. Consider the case where a 401x300 rectangle is to be filled with white pixels (pixel value zero on the Macintosh). This might happen, for example, when erasing a window. Furthermore, let's assume that the target bitmap has four bits per pixel, since that's somewhat tricker to handle than 8 bits per pixel. Figure 4 shows the subroutine that Executor dynamically generates to draw this rectangle on a Pentium.

```
loop: andl  $0xff,0x50(%edi)        ; clear leftmost 6 boundary pixels
      addl  $0x54,%edi              ; set up pointer for loop
      movl  $0x31,%ecx              ; set up loop counter
      rep
      stosl                         ; slam out 49 aligned longs
      andl  $0xffff0f00,0x0(%edi)   ; clear 3 right boundary pixels
      addl  $0x28,%edi              ; move to next row
      decl  %edx                    ; decrement # of rows left
      jne   loop                    ; continue looping if appropriate
      ret                           ; we're done!
```

Figure 4. Dynamically generated blitting code

This code, when called with the proper values in its input registers, will draw the entire rectangle. Note how the inner loop is merely a "`rep ; stosl`"...it doesn't get much more concise than that! The astute reader will know that on certain 80x86 processors "rep ; stosl" is not the fastest possible way to set a range of memory. This is true, but because our code generation is dynamic, in the future we can tailor the specific code sequence generated to the processor on which Executor is currently running. The blitter already does this when it needs to emit a byte swap; on the 80486 and up we use the "bswap" instruction, and on the 80386 (which doesn't support "bswap") we use a sequence of rotates.

One thing you may notice in this example is that the bit masks used to clear the boundary pixels look strange. They are actually correct, since 80x86 processors are little endian.

Unlike some processors, such as the 68040, the 80x86 instruction and data caches are always coherent. Consequently, no cache flushes need to be performed before the dynamically created code can be executed.

Figure 5 contains another example, this time drawn from a real application. The program "Globe", by Paul Mercer, draws a spinning globe

on the screen as fast as it can. Each "globe frame" is a 128x128 Pixmap. Here is the code that Executor generates and runs when Globe uses CopyBits to transfer one frame to the screen at 8 bits per pixel.

Again the inner loop is very tight, just a "rep ; movsl" this time.

```
loop:   movl   $0x20,%ecx       ; set up loop counter for 32 longs
    rep
    movsl                   ; copy one row (128 bytes)
    addl   $0xffffff00,%esi ; advance to previous src row
    addl   $0xfffffd00,%edi ; advance to previous dst row
    decl   %edx             ; decrement # of rows remaining
    jne    loop
    ret
```

Figure 5. Blitting code from Globe

Meta-Assembler

No matter how fast the generated code, if Executor spends too much time generating that code then any speedup will be negated by the increased time required for dynamic compilation. Consequently, the dynamic compilation from Region to 80x86 code needs to be fast. We solved this problem with a "meta-assembler" written in Perl.

Whereas an assembler tells a computer how to translate assembly instructions into machine code, our meta-assembler tells the computer how to generate tiny translators. These translators will then be used to translate pixel manipulation requests into machine code. Another way of looking at it is that the meta-assembler generates code that generates code. This meta-assembly process is done only once: when Executor is compiled.

The blitter operates on aligned longs in the destination bitmap. As the compilation engine strides through the scanline's start/stop pairs from left to right, it identifies which bits in each long are part of the Region and determines which of several pixel manipulation requests to issue to the tiny translators that were created by the meta-assembler.

- Some but not all bits in the current long are in the Region.
- All bits in the current long are in the Region.
- All bits in this long and the next long are in the Region.
- All bits in this long and the next two longs are in the Region.
- All bits in this long and the next three longs are in the Region.

- More than four contiguous longs are completely in the Region, and the number of longs equals 0 mod 4.

- More than four contiguous longs are completely in the Region, and the number of longs equals 1 mod 4.

- More than four contiguous longs are completely in the Region, and the number of longs equals 2 mod 4.

- More than four contiguous longs are completely in the Region, and the number of longs equals 3 mod 4.

The particular case encountered determines which function pointer to load from a lookup table corresponding to the current drawing mode. For example, the "patCopy" drawing mode has one table of function pointers, "patXor" another. There are also some special case tables for drawing patterns that are either all zero bits or all one bits.

The main blitter doesn't care what drawing mode is being used, since it does all mode-specific work through the supplied function pointer table.

Each function pointer points to a function that generates 80x86 code for the appropriate case. For example, one function generates code for a "patCopy" to three contiguous longs, one generates code for "patXor" only to certain specified bits within one long, etc.

The blitter compilation engine marches through the Region scanline from left to right, calling code generation functions as it goes. The generated code is accrued into a 32-byte aligned buffer on the stack. In this way, the blitter constructs a subroutine to draw the Region.

The compilation engine isn't very complicated. The tricky part is the numerous generation subroutines, which need to be fast since they are called so often and need to be easy to write since there are so many of them. For each drawing mode there's one for each case the compilation engine cares about. For pattern drawing modes, there are separate specialized subroutines for cases like patterns that can be entirely expressed in one 32-bit value ("short/narrow") patterns, patterns which can be expressed as one 32-bit value for each row, but which vary per row ("tall/narrow"), as well as "wide" variants of both. Beyond that, there are some versions specialized for 80486 and higher processors (which have the "bswap" instruction).

Generating fast and robust code generators is where the Perl meta-assembler comes into play.

The meta-assembler takes as input an assembly language template, and generates as output Pentium-scheduled assembly code that outputs an 80x86 binary for the input template. This process only takes place when Executor is compiled. Got it? This can be a little confusing, so a few examples are in order.

Here is perhaps the simplest template:

```
@meta copy_short_narrow_1
     movl  %eax,@param_offset@(%edi)
@endmeta
```

This template describes what should be done when the blitter wants to write one long to memory. The meta-assembler processes that into this 80x86 assembly code which is to be called by the blitter compilation engine:

```
      .align      4,0x90
_xdblt_copy_short_narrow_1:
      movw  $0x8789,(%edi)
      movl  %eax,2(%edi)
      addl  $6,%edi
      ret
```

The subroutine that the meta-assembler has produced above, when executed, will generate the movl instruction (i.e. the movl instruction in the template) followed by its argument. The meta-assembler has deduced that "movl" in the example template is 80x86 opcode 0x8789.

Let's take a look at a more complicated template. This template handles the case where we want to bitwise OR a pattern to the destination bitmap, and the number of longs to transfer equals zero mod 4 (e.g. if the blitter wants to OR 36 longs to memory):

```
@meta or_short_narrow_many_mod_0
      addl  $@param_offset@,%edi
     movl   $@param_l_cnt_div_4@,%ecx
1:    orl   %eax,(%edi)
      orl   %eax,4(%edi)
      orl   %eax,8(%edi)
      orl   %eax,12(%edi)
      addl  $16,%edi
      decl  %ecx
      jnz   1b
@lit  leal  (%eax,%edx,4),%ecx
@lit  addl  %ecx,edi_offset
@endmeta
```

The meta-assembler compiles that to this:

```
      .align      4,0x90
_xdblt_or_short_narrow_many_mod_0:
      movw  $0xC781,(%edi)
      movl  %eax,2(%edi)
      movl  $0x47090709,11(%edi)
      movb  $0xB9,6(%edi)
      movl  $0x8470904,15(%edi)
      movl  $0x754910C7,23(%edi)
      movl  $0x830C4709,19(%edi)
      movb  $0xEF,27(%edi)
      movl  %edx,%ecx
      shrl  $2,%ecx
      movl  %ecx,7(%edi)
      addl  $28,%edi
      leal  (%eax,%edx,4),%ecx
      addl  %ecx,edi_offset
      ret
```

This mechanically generated subroutine generates the executable 80x86 binary for the "or_short_narrow_many_mod_0" template. It gets called by the blitter compilation engine when it needs code to OR a bunch of longs to memory.

The output of the meta-assembler isn't meant for human consumption. As such, the output contains a hodge-podge of magic numbers (`0x47090709, 0xB9, 0x8470904,` etc.). These numbers are fixed machine code values corresponding to opcodes, constant operands, and other values.

Even though this subroutine is longer than the previous example, it still doesn't take very long to execute. Furthermore, it only gets called when the blitter has determined that many longs are to be ORed to memory, so the time taken actually blitting to memory will typically dwarf the time taken to execute these 15 code generation instructions.

The meta-assembler is a Perl script that works by running numerous syntactically modified versions of the assembly template through "gas", the GNU assembler, and examining the output bytes to discover which bits are fixed opcode bits and which bits correspond to operands. Once it has figured out what goes where, it generates 80x86 assembly code which writes out the constant bytes and computes and writes out the operand bytes. That code is run through a simple Pentium instruction scheduler and the meta-assembler is done. This entire process is, of course, done only once, when Executor is compiled.

A Portable Dynamically Recompiling Blitter

Although the meta-assembler-based blitter works only on 80x86 processors, Executor itself can run on non-Intel processors. On other CPUs (such as the 68040 used in the NeXTstation) Executor's blitter works somewhat differently.

The basic idea is still the same: translate Region scanlines into an efficient form once and then use that efficient form each time the scanline gets drawn. This time, however, the "efficient form" is processor independent, and the blitter is written entirely in C.

As is the case with the 80x86-specific blitter, the portable blitter compilation engine examines scanline start/stop pairs and identifies which of several cases is appropriate. One case is "output three longs", another is "output only certain pixels within the current long", and so on.

Like the 80x86-specific blitter, the particular case encountered determines which entry in a lookup table will be used. But there the similarity ends. The lookup tables contain pointers to C code labels[1] rather than to routines that generates 80x86 code on the fly.

[1] "What the heck is a pointer to a C code label?", you ask? gcc (the GNU C compiler) has a "pointer to label" extension to the C language which makes the statement "`&&my_label`" evaluate to a "`void *`" that points to the compiled code for "`my_label:`" within a C function. This, combined with gcc's "`goto void *`" extension, allows C programs to execute goto statements whose destinations are not known at compile time.

Each scanline gets translated into an array of opcodes for the "blitter opcode interpreter" (which will be described below). Each opcode is stored in one of these C structs:

```
struct
{
 /* Pointer to C code to handle
    this opcode. */
 const void *label;

 /* Offset into scanline */
 int32 offset;

 /* Extra operand with
    different uses. */
 int32 arg;
};
```

For example, consider the case where the blitter wants to write out five contiguous longs from a "simple" pattern starting 64 bytes into the current row. In this case, "label" would equal "&&copy_short_narrow_many_5", "offset" would equal 64, and "arg" would equal 5.

The Blitter Opcode Interpreter

The blitter opcode interpreter is machine generated C code created by a Perl script when Executor is compiled. That Perl script takes as input C code snippets that tell it how to handle particular drawing modes, and produces as output C code for an interpreter.

Here is the template taken as input by the Perl script for the "copy_short_narrow" case. This is the simple case where the pixels for the pattern being displayed can be stored entirely within one 32-bit long (for example, solid white or solid black).

```
begin_mode cpy_shrt_narrow max_unwrap
repeat      @dst@ = v;
mask        @dst@ = (@dst@ & ~arg)
                  | (v & arg);
end_mode
```

The "repeat" field tells the Perl script what C code to generate for the simple case where all pixels in the destination long are to be affected. The "mask" case tells it what to do when it must only modify certain bits in the target long and must leave others alone. Max_unwrap tells the Perl script to unroll the new blitting loop.

The generated interpreter takes as input an array of blitter opcode structs, which it then proceeds to interpret once for each row to be drawn.

Here is the section of the (machine-generated) interpreter that handles the copy_short_narrow cases. Remember that each "blitter opcode" is really just a pointer to one of these C labels. This code would get used when filling a rectangle with a solid color.

```
copy_short_narrow_mask:
 *dst = (*dst & ~arg) | (v & arg);
 JUMP_TO_NEXT;
copy_short_narrow_many_loop:
 dst += 8;
copy_short_narrow_many_8:
 dst[0] = v;
copy_short_narrow_many_7:
 dst[1] = v;
copy_short_narrow_many_6:
 dst[2] = v;
copy_short_narrow_many_5:
 dst[3] = v;
copy_short_narrow_many_4:
 dst[4] = v;
copy_short_narrow_many_3:
 dst[5] = v;
copy_short_narrow_many_2:
 dst[6] = v;
copy_short_narrow_many_1:
 dst[7] = v;
 if ((arg -= 8) > 0)
   goto copy_short_narrow_many_loop;
 JUMP_TO_NEXT;
```

Note how the inner blitting loop is "unwrapped" for speed. A blitter opcode would specify that 39 longs are to be output by making its "arg" field be 39 and the "label" field point to "copy_short_narrow_many_7", in the middle of the unwrapped loop (39 mod 8 equals 7). The interpreter would jump there and loop until all of the pixels had been written out, at 32 bytes per loop iteration. This is very fast, especially for portable code.

Of course, if any other pixels needed to be drawn, there would be additional blitter opcode structs telling the interpreter what to do. The interpreter dispatches to the next opcode by executing the "JUMP_TO_NEXT" macro, which automatically uses GCC's "goto void *" extension to "goto" the C label that handles the next opcode.

Development Tools

Free Software

It is true that ARDI has a very tight R&D budget, but we really don't skimp on the tools that we use to build Executor. We use free software to develop Executor because we like to push the tools that we use very hard and the only way we can do that and still sleep at night is when we know that if we find bugs in our tools that they can be fixed quickly. With free software the worst case is to fix bugs ourselves, and that worst case is actually much better than the average case with non-free software where you report a bug and pray for a patch. In reality it's rare that we even have to resort to the worst case since bugs reported are often fixed in less than a day.

GCC

GCC is the Free Software Foundation's C compiler. It produces good code and has a powerful inline assembly syntax that allows optimization to be done on the expressions in the inline assembly without the optimization ruining the assembly you've written.

Another handy GCC extension is "`typeof`" which can be used in macros to cast a value to the type of a different value. The combination of powerful inline assembly and typeof allows us to have efficient macros that swap bytes in a 16 bit or 32 bit quantity. Since the Mac and PC are of different endianness, quick byte swapping routines are very important.

As mentioned above in our synthetic CPU and portable blitter descriptions, we also use GCC's ability to take the address of a label and store it in a variable so that we can produce our own threaded code on the fly.

Hacked GCC

Because the source to GCC is available, it is possible, although not necessarily advisable, to hack in custom extensions. At ARDI we've done this twice in the past. At one time we used a set of locally written modifications to support the pascal keyword so that we could automatically call functions using Pascal calling conventions. At the same time we also supported '`1234`' (i.e. the ability to construct a 32-bit quantity out of four character constants inside apostrophes). Eventually we decided that we didn't get enough benefit from these extensions to make it worth patching new versions of GCC as they came out.

The other time we modified GCC was when we were porting Executor to DEC's Alpha processor. We were doing this under OSF/1 which uses 64-bit pointers. Since Executor needs to use the same internal representation that Macs use, we wanted a way to easily write 32-bit pointers to memory in such a way that they would be extended to 64-bits when they were read into a register for use. To do this we made GCC support "pointer bit fields", a logical extension that allowed bit-field notation to be used when specifying pointers. At that time we didn't have a resident GCC expert, so we were lucky that such modifications basically consisted in taking out a few checks that disallowed such constructs. Once those checks were removed, pointer bit-fields, "just worked".

DJGPP

DJGPP is DJ Delorie's (see `http://www.delorie.com`) port of GCC to MSDOS. It allows DOS users to compile UNIX programs under DOS and to run them with little or no modification. DJGPP is GCC and associated development tools with a special UNIX like C-library and a "DOS Extender". DOS extenders are used to combat OS inferiority. DOS is a 16-bit OS, whereas most relatively modern OSes are 32-bit. DOS extenders allow 32-bit programs to run under DOS. Executor is one such program. In fact, we use the djgpp libraries and DOS extender but we don't actually use the DOS port of GCC, because we don't like DOS. We like Linux and GCC is well structured so we can do cross-compilation and cross-linking with the djgpp libraries and build our DOS product under Linux. We completely compile the DOS version of Executor under Linux. We then copy the new Executor binary to a DOS partition, reboot to DOS, test Executor and then get the heck out of DOS. Time spent using Executor is more like a Mac than it is like DOS.

Debugging Tools
Internally we have many debugging tools to help us figure out why an application may die or misbehave under Executor.
More Free Software

GDB in General

Almost all of our debugging is done under the GDB debugger. As with GCC, we're not using GDB because it's the free debugger; we're using the free debugger because it's GDB. GDB is quite powerful.

Whenever we find that a given application fails under Executor, we try to reproduce the failure under Linux. Debugging on a system that has complete memory protection and pre-emptive multi-tasking means that your system stays up even when your application crashes. There's also no need to worry that when a program is misbehaving that it's subtly corrupting other programs on the system.
hardware watch points
Beyond the features that are handed to us due to the underlying robustness of the OS, GDB also supports hardware watch points, at least on 80x86 based PCs. "80x86"s have the ability to use hardware to watch a small set of memory locations to see when they change. Since the checking is done by hardware, the program runs at full speed until the memory location is modified, at which point the debugger stops, tells us which instruction modified which memory address and what the old and new values are for that address.

As an example, assume we want to know when the low-memory global `TheMenu` is changing, here is how it might look under GDB:

```
(gdb) watch TheMenu
Hardware watchpoint 1: TheMenu
(gdb) c
Continuing.
Hardware watchpoint 1: TheMenu

Old value = 0
New value = 768
C_HiliteMenu (mid=3) at menu.c:877
(gdb) swap16 768
$2 = 0x3
(gdb) c
Continuing.
Hardware watchpoint 1: TheMenu

Old value = 768
New value = 0
C_HiliteMenu (mid=0) at menu.c:877
(gdb) delete 3
(gdb) c
Continuing.
```

At the first `(gdb)` prompt above, we tell GDB that we want to be alerted whenever the expression "TheMenu" changes. GDB is clever enough to realize that it can watch that expression with a hardware watchpoint, so it assigns watchpoint 1 to the task. We then continue, which allows Executor to continue running

whatever program it was already running.[2]

Eventually, when the menu bar was accessed, GDB told us that TheMenu had changed from 0 to 768. 768 may sound like a weird value for TheMenu to take, but this is on a byte swapped machine, so we need to swap that 16-bit value to see what the TheMenu would look like to a Mac program and we find that it's 3, a sane value for TheMenu, after all. We let the program continue and later TheMenu is changed back to zero.

You can't see it, but in another window the source to Executor is displayed so that we are automatically shown the 877th line of menu.c when GDB's watch point triggers there.

The argument to the watch command is an arbitrary expression, so it is possible to watch for much more complex changes than our example demonstrated. Although only relatively simple watchpoints will be handled by hardware watchpoints, the others will be handled by software watchpoints which are much slower.

[2]I actually set this watchpoint in the session of Executor that I am using to run Word 5.1 for the Macintosh to compose this document (Executor/Linux on a 90 MHz Pentium).

Hacked GDB

Unlike GCC, where we made local modifications and then, upon reflection, threw them out, we have made a slight change to GDB that is a big win for debugging Executor (and Mac programs running under Executor) on PCs. GDB always knows how to disassemble the object code that it's running, and GDB is available for many architectures, so we modified GDB so that on the 80x86 we can do both 80x86 disassembly and 680x0 disassembly. That allows us to look at sections of memory within our emulator and see what 680x0 code is there.

In the example below, Executor is running the game Risk, when we interrupt Executor and then tell GDB to break in the routine alinehandler. We then continue until alinehandler is hit. We then disassemble, in 680x0 format, the first nine instructions at the location from which alinehandler was dispatched. After doing that we disassemble in 80x86 format the first nine instructions of alinehandler itself.

```
(gdb) b alinehandler
Breakpoint 6 at 0x17ce2d: file executor.c,
line 369.
(gdb) c
Continuing.

Breakpoint 6, alinehandler (pc=3652006,
ignored=0x0) at executor.c:369
(gdb) set m68k
(gdb) x/9i pc
0x37b9a6 :  _SystemTask
0x37b9a8 :  clrw sp@-
0x37b9aa :  movew #-1,sp@-
0x37b9ae :  pea a5@(-27598)
0x37b9b2 :  _GetNextEvent
0x37b9b4 :  moveb sp@+,d0
0x37b9b6 :  tstb d0
0x37b9b8 :  beqw 0x37ba0e <end+667542>
0x37b9bc :  movew a5@(-27598),d0
(gdb) set m68k off
(gdb) x/9i alinehandler
<alinehandler>:         pushl  %ebp
<alinehandler+1>: movl   %esp,%ebp
<alinehandler+3>: subl   $0x28,%esp
<alinehandler+6>: pushl  %esi
<alinehandler+7>: pushl  %ebx
<alinehandler+8>: jmp    0x17ce10
<alinehandler+48>
<alinehandler+10>:      nop
<alinehandler+11>:      nop
<alinehandler+12>:      nop
```

Being able to disassemble 680x0 code on the 80x86 required us to change approximately 50 source lines of GDB (remember, the 680x0 disassembly code was already present for use in GDB on 680x0 machines). We also added a set of tables so that a-line traps and low-memory globals are displayed by name, rather than by number.

Although our special circumstances led us to modify the GDB source code, GDB is customizable out of the box. We've defined a handful of macros that automate debugging tasks. Figure 6 is a macro that crawls through the stack in mac space.

For comparison, Figure 7 is what GDB produces when backtracking code that is compiled with GDB debugging symbols.

```
define macktrace
 set $_fp = cpu_state.regs[14].ul.n + 0
 silentswap32 (((uint32*)$_fp)[1]+0)
 set $_pc = $_val + 0
 silentswap32 (((uint32*)$_fp)[0]+0)
 set $_fp = $_val + 0
 while $_fp > 100 && $_fp < 30000000
   set $_start = (long) $_pc + 0
   while $_start > (long)&end && *(uint16 *)$_start != 0x564E
      set $_start = $_start - 2
   end
   printf "func=0x%lX, ret=0x%lX, fp=0x%lX, args=0x%02X%02X%02X%02X 0x%02X%02X%02X%02X
0x%02X%02X%02X%02X\n",\
      $_start, $_pc, $_fp,\
      ((uint8 *)$_fp)[8], ((uint8 *)$_fp)[9], ((uint8 *)$_fp)[10],\
      ((uint8 *)$_fp)[11], ((uint8 *)$_fp)[12], ((uint8 *)$_fp)[13],\
      ((uint8 *)$_fp)[14], ((uint8 *)$_fp)[15], ((uint8 *)$_fp)[16],\
      ((uint8 *)$_fp)[17], ((uint8 *)$_fp)[18], ((uint8 *)$_fp)[19]
   silentswap32 ((uint32*)$_fp)[1]+0
   set $_pc = $_val + 0
   silentswap32 ((uint32*)$_fp)[0]+0
   set $_fp = $_val + 0
 end
end
(gdb) macktrace
func=0x3824F8, ret=0x38250E, fp=0xB28E3C, args=0x00B2E852 0x000300B2 0x8E580037
func=0x37A2BE, ret=0x37A3A6, fp=0xB28E4A, args=0x0037BA12 0x000000B2 0x8F840037
func=0x37AECE, ret=0x37AFF0, fp=0xB28E58, args=0x0001002E 0xE0BC0000 0x00010035
func=0x379D58, ret=0x379E0C, fp=0xB28F84, args=0x000100B2 0x8F9200B2 0x8F9A0000
```

Figure 6. Macktrace Definition and Example

```
(gdb) backtrace
#0  C_SysBeep (i=10) at osutil.c:837
#1  0x18934d in PascalToCCall (ignoreme=2271560241, infop=0x29faa4)
   at emutrap.c:94
#2  0x17d0c9 in alinehandler (pc=3661160, ignored=0x0)
   at executor.c:399
#3  0x1c1b85 in trap_direct (trap_number=10, exception_pc=3661160,
   exception_address=0) at trap.c:201
#4  0x197cfc in S68K_HANDLE_0x00B5 () at syn68k.c:1038
#5  0x196067 in interpret_code (start_code=0x2df6c4) at syn68k.c:587
#6  0x12d476 in beginexecutingat (startpc=11730018)
   at launch.c:328
#7  0x12e1ce in launchchain (fName=0x2b53f8 "\004Risk", vRefNum=-32717,
   resetmemory=1 '\001') at launch.c:575
#8  0x12f6e0 in Launch (
   fName_arg=0x910 "\004Riskutor", '˜' <repeats 27 times>, vRefNum_arg=-32717)
   at launch.c:1142
#9  0x17e1f7 in executor_main ()
   at executor.c:589
#10 0x13371a in main (argc=2, argv=0xbffffa04)
   at main.c:2112
```

Figure 7. GDB backtrace

As you might guess, this disparity of information makes it much easier for for us to track down bugs in our own code then finding bizarre incompatibilities in the code that is being run under the emulator.

Disassembler

Since GDB already knows how to disassemble 680x0 code it was possible to write a driver for GDB so that it can disassemble Mac programs. The driver is about 1,000 lines of C code, with another 500 lines describing the low-memory globals. Basically the driver knows about CODE resources and how intersegment jumps work. GDB normally doesn't produce labels for jump targets or the beginning of subroutines, so the driver adds those too, to make the output that much easier to read.11

Run-time Aids

Because we're using our own set of OS and Toolbox routines, we can add code that is conditionally compiled into debug versions of Executor that can provide still more information than GDB or GDB macros can.

Debugtable, Debugnumber

Our A-line trap handler has a table, known as debugtable, of 4096 32-bit ints that it updates each time a trap is taken. Each time alinehandler is called, a variable known as "debugnumber" is incremented and then the value of debugnumber is stored in the slot in debugtable corresponding to the aline trap that was called. This allows us to see both what traps were recently executed and a complete list of every trap that an application makes, no matter how long the application has run.

This scheme has its drawbacks. Traps that are dispatched via selectors are all lumped together. Traps whose addresses are taken and then are called by jumps through the address don't show up in debugtable. Although debugtable and debugnumber are perhaps the least sophisticated portion of Executor, they're still quite handy, since a visual inspection of the last 100 traps made before an application died often gives a good idea of where to start looking for the source of the incompatibility.

XX_slam

In the course of developing Executor, we did a major rewrite of our memory manager and our TextEdit replacement. In both cases it's not enough to just implement the APIs that are defined in Inside Macintosh, we also have to duplicate the in-memory data structures so that programs which count on them will run properly. To help us verify that we weren't adding new bugs when we rewrote those subsystems we added routines that would consistency check the data structures that each of those subsystems support.

Because these consistency checks are thorough but time consuming, we call them "slams", and by default they are not enabled, even in debugging versions of Executor. When they are enabled, the data structures for each subsystem are slammed at the entry to a call that might modify one of the data structures and the data structure is slammed once again on exit of the routine. We can turn them on at run-time either by using a command line option when Executor is started or by using GDB to enable the slamming. This is something we should have done for all of Executor's subsystems from day one, since it's ever so helpful to be told that going into routine XXX, the heap was fine, but coming out the heap was corrupted.

Image Viewer

Reading disassembled code is much easier than staring at hex numbers. Similarly, being able to view a portion of memory as some sort of PixMap (assuming that the memory really is a bit image) is also better than staring at a bunch of hex numbers. When we build Executor for X-Windows, we also build an image server that uses UNIX interprocess communication to communicate with the process being debugged under GDB. This allows us to monitor offscreen graphics, which can be very important when an application makes many graphics calls and eventually an abomination is drawn on the screen instead of what should have been drawn.

Our debugging arsenal includes other, more prosaic, tools. In fact, our debugging environment encourages the development of new tools, because it's so easy to leverage existing tools into new tools and even write new tools from scratch.

Future Plans

Much of VCPU, a successor to Syn68k, has already been written. VCPU performs many optimizations that Syn68k does not, including improved register allocation, dead subregister elimination, opcode "widening", and moving work outside of loops. VCPU has a clean high-level syntax for specifying both front ends and back ends, allowing it to dynamically compile both PowerPC and m68k binaries on any architecture we decide to support.

Although we don't explicitly mention it, the graphic subsystem one layer above the blitter already has hooks in it to allow use of graphics accelerators, where present. We plan a native port to Win32 and OS/2 and those ports should be able to use fancier graphic subsystems and also make use of the underlying network APIs.

Currently INITs and CDEVs do not run under Executor, but the same mechanisms that allow applications to run can also allow INITs and CDEVs to run. QuickTime and ATM will both be high priorities after Executor 2 ships.

We will also be developing compiler tools that will allow ISVs to natively compile CPU specific routines to be used when their applications are run under Executor. Executor already uses such gateways internally.

Already, multiple simultaneous instances of Executor can be run under NEXTSTEP and Linux (and to a lesser extent under Windows '95). Currently only Executor/NEXTSTEP handles PICT pasteboard cutting and pasting from one instantiation of Executor to another, and no versions of Executor do enough file locking to allow concurrent access of the same HFS volumes at once. This needs to be fixed, since either through shared text segments under UNIX and UNIX like operating systems or through DLLs under Microsoft operating systems, it can be made fairly efficient to run multiple instances of Executor simultaneously. When that is done, each instance of Executor has its own address space and is automatically scheduled by the underlying operating system scheduler. That means that Executor "inherits" memory-protection and pre-emptive multi-tasking from the underlying core operating system.

By properly exploiting this inheritance it should be possible to provide an environment that allows well-behaved Mac applications to run efficiently under a variety of PC operating systems with automatic protection from non-well-behaved applications.

One interesting variant on this theme would be to use Linux as the core OS, but to hide it from the end-user, for a net result of an 80x86 box that boots an efficient, robust MacOS-like environment.