The Technology of Emulation: 68K on PowerPC

(revised 94 June 20)

by Tom Pittman

Abstract

The technological aspects of processor emulation in software are discussed with particular reference to efficient emulation of the 68000 family on the PowerPC 601. Special attention is given to the instruction fetch & decode loop and to condition code handling. A skeleton emulator is presented, with a statistical performance comparison to Apple's.

Introduction

It is well known that virtual machine (VM) emulation generally costs about one order of magnitude in performance, as compared to native-code execution. The published figures for Apple's first-generation Power Macintosh are quite consistent with this. In this paper we discuss how to get the best performance in VM emulation, using the PowerPC as host and the 68000 as target. We also look briefly at the way Apple did it and consider how they might improve it in future products.

Elements of Emulation

An interpreter consists of three essential parts. First is the native representation of the VM state, including any registers that are accessible to the emulated VM code. The 68000 family contains eight general data registers, eight address registers (one of which is defined in the hardware as a stack pointer), a program counter and a machine status register (mostly, for our purposes, the condition codes) [1]. There are other registers controlling things like virtual memory, but programmers are discouraged from accessing them directly so they can be readily ignored. Similarly, some 68040 processors contain on-chip floating-point hardware, but Apple explicitly declined to emulate this functionality, and it only complicates our analysis without adding any understanding; we ignore this also. The 601 processor [2] has 32 general-purpose registers, of which all but perhaps three or four are available for the emulator. The VM state is therefore easily consigned entirely to 18 GP registers, leaving another ten for intermediate values required for emulation.

The second essential part of the VM interpreter is the execution engine, a very tight loop that fetches each successive instruction from the VM code and dispatches execution to the specific code to emulate that instruction.

The third part is the specific routines for each supported VM instruction. Except where the two architectures differ in their results and/or side effects, the actual code in the third part will closely resemble the cognate code of the emulated machine. There is also a lot of repetitious similarity in this third part, so apart from a few representative examples here, we spend most of our time looking at the critical second part, where most of the emulation overhead is spent.

The Execution Engine

The execution engine, also sometimes known as the Fetch Loop, may be further subdivided into five elements:
1. Fetching the VM instruction,
2. Incrementing the VM program counter,
3. Extracting the operation code (normally abbreviated "opcode") from the instruction,
4. Using it to select and jump to the emulation code for that instruction, and
5. Returning to step 1 in the loop.

Depending on the time/space trade-off desired, the execution engine may also decode the operand

addressing modes and possibly fetch operands before jumping to the specific emulation code.
In our particular context, we observe that each 68000 instruction is always some multiple of two bytes, and the first two-byte piece uniquely defines both the operation to be performed and (with two infrequent exceptions) how many more bytes of operand address are required by this instruction. This forms a natural bound for step 1: the VM fetch should get two bytes at a time. Both the 68000 and the 601 — indeed most modern processors — can combine the fetch and increment of steps 1 and 2 in a single instruction. In the 68000, it is a post-increment operation, that is to say, the fetch occurs on the unincremented address, then the register (whether the program counter in the case of the hardware instruction fetch, or the address register in the auto-increment addressing modes) is updated with the incremented address. In the 601 and other PowerPC processors, the reverse is true (at least of registers used in the auto-increment mode): the increment is added first, then the operand is accessed with the new address. Therefore to combine the fetch and increment, the virtual program counter must be emulated by an actual register that is always 2 less than the virtual PC. This decision will also impact branch, jump, and return instructions, but not by much.

We can further improve performance by prefetching the next opcode before decoding and executing the current one. Thus the memory latency for fetching each VM instruction (which is often not in the primary on-chip cache) can be overlapped with the execution of the previous instruction. This may cost extra time in the case of short branches and returns, which must discard the prefetched but unused opcode following the current instruction, and some extra complexity (but no extra cost) for all sequence-control instructions in the VM, which must initiate a prefetch on the target address. However, this is no more novel than instruction pipelining, which computer hardware has been doing for years.

Table 1. The 16 Major Opcodes of the 68000

Bits 15–12		General Description
0000	A1	Immediate operations
0001	A2	Move Byte
0010	A2	Move Long
0011	A2	Move Short
0100	A1	Miscellaneous
0101	A1	Add/Sub Quick & CC
0110	D	Branch
0111	D	Move Quick
1000	A1	Or & Divide
1001	A1	Subtract
1010	–	Illegal (A-Traps)
1011	A1	Compare & Xor
1100	A1	And & Multiply
1101	A1	Add
1110	A1	Shift & Rotate
1111	–	Illegal (Floating-point)

Key to Instruction formats:

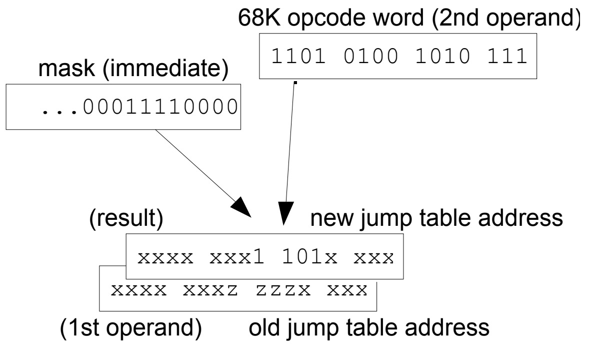
–	Undefined
A1	Single-Operand Address Mode in 5–0
A2	Double-Operand Address Modes in 11–0
D	Data or Offset in 7–0

Extracting the opcode can be simple and fast, or complex and relatively slow (but compact). Let’s consider the latter first. The major opcode of the 68000 family instruction set is the most significant four bits (see Table 1). This also selects one of two subdivisions of the remaining 12 bits. In two cases (opcodes 0110 and 0111), the bits are divided 4+8, with the four-bit piece being used to select a register or a condition to test (sort of a minor opcode, but not quite), while the lower byte is immediate data. In most of the other cases the bits are divided into two 6-bit register and addressing mode selectors, where one or both of these addressing mode fields often doubles for a minor opcode — usually as determined by the upper four bits. Two of the major opcodes were originally reserved for linkage to co-processors, but Apple subverted one of them for system calls. Both are defined as illegal opcodes with their own interrupt vector address, and are therefore not proper instructions. Many of the other opcodes are also defined as illegal, or else redefined as some special instructions. Table 1 ignores these special instructions, but of course the emulator must deal with them.

The compact (and more complex) instruction decode would extract the four-bit major opcode and use it to choose from a table of 16 instruction formats, which further select the subroutines to decode the addressing modes (and special instructions where they occur). In the 601, a single instruction can extract and position the four-bit major opcode piece for indexing the major decode table; another instruction fetches the entry from that table. A table entry typically would be the address of the code to do the operation, and that code normally begins with either a subroutine call or inline code to decode the addressing mode and fetch the operands. Memory data fetches cost a minimum one-cycle penalty even if the data is in the cache (much more if it is not, but for the interpreter’s decode table we can generally assume the best case). So the conventional approach here turns out to be less efficient than we might hope for.

Branch instructions in the 601 are free if we can set up the branch predictor well enough in advance, so it makes more sense to make the decode table entries big enough to hold most of the code (or at least another jump) and simply jump to the table entry itself. The 601 lacks an indexed branch like we have in the 68K and most older architectures, so it takes two more instructions to build the jump address: one adds the four-bit opcode (already shifted left to accomodate the table entry size) to the base table address, and the second extra instruction copies the resulting table entry address to one of two special PowerPC registers that can hold computed jump addresses. The add instruction can be eliminated if the table is forced to a power-of-two boundary, so that the indexed part of the table address can be mask-inserted instead of added. This works because the 601 has an atomic shift-with-masked-insert instruction that takes a register value (the fetched opcode in our case), rotates it left or right, then replaces some number of contiguous bits in another register with the corresponding bits of the rotated value, as shown in Figure 1. So we merely keep the address of the table in a dedicated register, then shift the opcode into position and jam it into the middle bits of this register and jump to it. Jumping into the table like this results in a net performance gain over the conventional approach in the 601.

Figure 1. The PowerPC **`rlwimi`** instruction



Upon arrival in each of the sixteen decoded routines, a sequence of extract-and-jump instructions like the one just described for the major opcode now decodes one or both minor opcodes as appropriate, and only then can the specified operation take place. Branch prediction works to our advantage, so the addressing mode decode can be relegated to a small number of subroutines: the jump to the subroutine and the return are essentially free. The extraction and table fetch (or jump) in each case, however, are not free; they cost three or four instruction cycles each.

At the cost of a larger major opcode table, we can implement the simple and fast method, performing all the decoding at once. The absolutely fastest 68000 instruction decode requires at least a megabyte of table space, so that each possible 16-bit instruction has 16 bytes (four instructions) to attempt execution. A few would take less, many would take more, but most could be accomplished in that space. When more is needed, the last instruction in the table entry is a jump to some excess-codecontinuation area, which is as large as needed in each case. Making the jump instruction the last of four allows the branch prediction time to look ahead, so the jump is free. According to the new PowerPC volume of *Inside Macintosh* [3], Apple’s emulator uses two instructions for each table entry (for a resulting half-megabyte table), which almost always requires a jump to the excess code. The very minimum table entry would be a single 4-byte instruction (a jump), but the extra instruction in Apple's implementation permits some customizing based on the decode, before jumping to common code in the overflow area.

Listing 1 shows the code for the fast fetch loop (using Motorola mnemonics). Note that each table entry jumps directly to the front of the loop (label FetchLoop) on completion, eliminating the need for a branch at the end of the loop. Execution time for this is a nominal 3 cycles (actually 5 due to scheduling latency, but more on this later):

Listing 1. The Emulator Fetch Loop

```
FetchLoop:
       rlwimi    tablead,prefetch,16,12,27     ; build table address
       mtspr     ctr,tablead                   ; prepare for jump
       lhau      prefetch,2(vpc)               ; prefetch next opcode now
       bctr                                    ; jump to table
```

Condition Codes

One of the problems we must deal with in instruction emulation is the 68000 condition codes (CCs). The PowerPC uses a different model for condition code testing, so a poorly crafted implementation of CC handling could put a costly dent in our emulation speed (my first cut at 68000 emulation was almost twice as slow as the best performance reported here). The simplest and fastest way to deal with the matter is to procrastinate as much as possible. Ideally, we would not place the bits into the virtual processor state register until some VM instruction needs it. Most 68K instructions that alter the CCs are followed by other instructions that replace that CCs with something else — without ever looking at the first result. The goal is to do as little as possible to save this part of the state until it is really needed. It helps that many 601 instructions can optionally set the PowerPC equivalent of the condition codes or not; we choose to not set them when the VM instruction would not.

The 68000 instruction set may alter any or all five CCs, or none at all. The X bit (if set) follows the C bit. There are eight different ways the CCs can be affected, as listed in Table 2. The notation in Table 2 is derived from the Motorola 68000 reference manual [1], but if you think of each zero in the table as just another star, it simplifies to four classes. The X bit can be isolated by copying the 601 carry flag to a separate place after adding, subtracting, and those shift operations that affect it. The bit operations introduce a rather strange anomaly, where Z and N can be both true at once. Luckily, the 601 is also capable of such an anomaly, by explicit programming of the CR register bits. The designers of the 68000 apparently thought the Carry bit in the CCs should mean "borrow" in the case of subtraction and compare, while in the 601 it is a true carry out of both addition and subtraction, which half the time is exactly the opposite of the 68000. This is readily corrected by inverting the bit after emulating subtraction and compare instructions.

Table 2. Distinct Condition Code Combinations of the 68000

XNZVC	
`*****`	Arithmetic: X follows C
`-****`	Compare
`***0*`	Shift/Rotate: X follows C if shift count ≠ 0
`-**0*`	Shift/Rotate without extend
`-***0`	Divide
`-**00`	Logic (including Move) & multiply
`--*--`	Bit operation
`-----`	Condition Codes are not changed

Key:

`-`	No change
`0`	Force to zero
`*`	Altered

Conditional branches that test for various combinations of the 68K CCs may require emulation by means of a sequence of two or more 601 branch instructions, but the most common 68000 CC tests are emulated in a single 601 branch-on-condition test. Thus we can preserve the conditions in the 601 format (in the CR register) after arithmetic, logic and compare instructions, and test the 601 CR register when emulating the 68000 conditional branch. The 68000 Scc instruction, although less frequent, can use the same test algorithm as the conditional branch. Copying or altering the CCs directly is rare enough that the extra cost to convert between the 601 model in the CR register and the 68000 model in the VM processor state register will not seriously impact total execution performance. Similarly, the only use of the X Condition Code bit in the 68000 is for rarely-used extended shifts and add/subtract instructions, so the extra cost of converting a saved copy of X is justifiable.

The emulation of arithmetic operations and compare instructions execute in the interpreter code with the result recorded in CR0 and the carry bit of the XER register. The arithmetic operations use a second instruction to capture X as a copy of the carry bit from XER. Shift and rotate instructions are similar to arithmetic, but being less frequent, we can tolerate a few extra instructions to cope with the differences in X bit management.

The net burden CC processing places on the emulator is an occasional one or two instruction cycles that would otherwise be unnecessary. Add operations require one extra instruction to rescue the X bit from the C; subtract and compare also require two instructions to invert the carry. Move operations (except MOVA) must be done with 601 arithmetic instructions, adding 0 so that the condition codes are properly set. Logic operations are comparable. Conditional branches typically involve no overhead (the flag tests are essentially free because of branch prediction) unless testing the Carry bit in unsigned compare results, which costs two or three instructions to set up. All other instructions either do not involve the CCs at all, or are so infrequent as to have little impact on execution time.

Addressing Modes

The 68000 has ten distinct operand addressing modes; the 68040 has a few more, but compatibility with Classic Macs (which were all 68000-based) tends to diminish their importance in commercial software. Some of these modes are restricted to prevent their use as destinations, but this is of little concern to us in emulation (Apple, with a goal of emulating faults as well as normal operation, must generate corresponding exceptions; however, most of these decisions can be made as part of the instruction decode). Only the move instructions have full generality in source and destination operands; the two-operand arithmetic and logic instructions must have at least one operand in a register.

The most complex addressing modes require significant calculation to compute the effective address, which must be loaded into a separate 601 register for execution. Except for the two indexed modes (codes 110 and 111011), all can be calculated in a single step: recall that the address offset has already been prefetched in the execution loop. Decoding the variety of addressing modes from the second word of the two indexed modes requires another costly extraction and table-lookup jump; fortunately these are infrequent. After the operand address has been computed, the operand itself can be fetched. A small number of continuation routines capture most of the common code that cannot fit in the four-instruction decode table entry.

Listing 2 shows representative code to fetch a longword operand for each (68000 only) addressing mode; memory stores and other operand sizes are obvious extensions of these. Note that each of these fragments of code would normally be encoded inline in their respective parts of the instruction decode table (or an extension of it), as shown in the decode examples in the next section, and not preserved in a separate table as shown in Listing 2.

The register index table consists of two instructions in each of 32 entries: a register copy or sign-extend, followed by a **bclr** (subroutine return). In the 68040 the indexed addressing format encodes many complex modes, so the table must be expanded to 256 entries to fully decode both the index register and the additional mode bits. While important for completeness, it is not particularly relevant to our discussion here.

Listing 2. Addressing Mode Calculation Fragments

```
RegDirect:                                          ; 000/001
       ori      opreg,<reg>,0                   ; just get the register
RegIndirect:                                        ; 010
       lwz      opreg,0(<reg>)                  ; load using address in reg
PostIncrement:                                      ; 011
       lwz      opreg,0(<reg>)                  ; load using addr in reg,
       addi     <reg>,<reg>,4                   ; then increment reg
PreDecrement:                                       ; 100
       lwzu     opreg,-4(<reg>)                 ; load with predecrement
RegOffset:                                          ; 101
       lwzx     opreg,<reg>,prefetch            ; load @(reg+disp)
       lhau     prefetch,2(vpc)                 ; prefetch next opcode
RegIndexed:                                         ; 110
       rlwimi   tabreg,prefetch,24,24,28        ; build table address
       mtspr    ctr,tabreg                      ; prepare for jump
       extsb    opreg,prefetch                  ; this is the offset part
       lhau     prefetch,2(vpc)                 ; prefetch next opcode now
       bctrl                                    ; jump to table with link
       add      temp,temp,opreg                 ; add index part + offset
       lwzx     opreg,<reg>,temp                ; load it
AbsShort:                                           ; 111000
       lwz      opreg,0(prefetch)               ; load using displacement
       lhau     prefetch,2(vpc)                 ; prefetch next opcode
AbsLong:                                            ; 111001
       lwz      temp,0(vpc)                     ; refetch address long
       lhau     prefetch,4(vpc)                 ; prefetch next opcode
       lwz      opreg,0(temp)                   ; load using fetched addr
PCoffset:                                           ; 111010
       lwzx     opreg,vpc,prefetch              ; load @(pc+disp)
       lhau     prefetch,2(vpc)                 ; prefetch next opcode
PCindexed:                                          ; 111011
       rlwimi   tabreg,prefetch,24,24,28        ; build table address
       mtspr    ctr,tabreg                      ; prepare for jump
       extsb    opreg,prefetch                  ; this is the offset part
       bctrl                                    ; jump to table with link
       add      temp,temp,opreg                 ; add index part + offset
       lwzx     opreg,vpc,temp                  ; load it
       lhau     prefetch,2(vpc)                 ; prefetch next opcode
Immediate:                                          ; 111100
       lwz      opreg,0(vpc)                    ; refetch long data
       lhau     prefetch,4(vpc)                 ; prefetch next opcode
```

Detailed Emulation Code Examples

We now consider some examples of emulation code for common VM instructions. Listing 3 shows the code fragments. The **MOVA** instruction does not alter any condition codes, so it is quite simple. Each addressing mode is separately decoded, but this example loads using the RegOffset mode as described above (2 cycles). The **MOVQ** instruction is fully decoded in the jump, so its emulation code is one instruction to load the specified constant into the specified virtual data register and set the CCs (1 cycle). Motorola overloaded the **MOV** mnemonic for a variety of quite different operations, which are all distinguished in the instruction decode. We illustrate a register store short with post-increment (4 cycles).

For an example of arithmetic, we show an **ADD** instruction that uses the register indirect addressing mode (3 cycles, plus memory latency). Another example of arithmetic, the **SUBQ** instruction directly decrements a data register. If the destination were an address register, the CCs would be unaffected. Note that this is fully decoded, including the amount to be subtracted (3 cycles). In the **CMPI** instruction we compare a data register to an immediate byte value (6 cycles).
The **BMI** instruction is an example of a conditional branch short (nominal 0 or 1 cycles). Finally, a slightly more complicated subroutine call, in which the **JSR** instruction also requires the return address to be pushed on the stack (still only 4 cycles).

Listing 3. Typical Emulation Code Examples

```
MOVA    lwzx     <dest>,<reg>,prefetch         ; load @(reg+disp)
        lhau     prefetch,2(vpc)               ; prefetch next opcode
        b        FetchLoop                     ; go back for next

MOVQ    addic.   <dest>,zeros,<const>          ; const into reg, set CC
        b        FetchLoop                     ; go back for next

MOV     extsh    temp,<src>                    ; get the register
        addic.   temp,temp,0                   ; clear carry and set CC
        sth      temp,0(<reg>)                 ; store using addr in reg,
        addi     <reg>,<reg>,2                 ; then increment reg
        b        FetchLoop                     ; go back for next

ADD     lwz      temp,0(<src>)                 ; load 2nd operand
        addc.    <reg>,<reg>,temp              ; put result back, set CC
        adde     xbit,zeros,zeros              ; copy out X bit
        b        FetchLoop                     ; go back for next

SUBQ    addic.   <reg>,<reg>,<-imm>            ; do it, set CC
        subfe    xbit,zeros,zeros              ; recover inverted X bit
        addc     temp,xbit,xbit                ; copy inverted X back to C
        b        FetchLoop                     ; go back for next

CMPI    andi     temp,<reg>,255                ; get reg (low byte only)
        andi     prefetch,prefetch,255         ; 2nd operand in prefetch
        subc.    temp,temp,prefetch            ; discard result, keep CC
        lhau     prefetch,2(vpc)               ; start another prefetch
        subfe    temp,zeros,zeros              ; invert the carry
        addc     temp,temp,temp                ; copy it back to CA in XER
        b        FetchLoop                     ; go back for next

BMI     bc       4,0,FetchLoop                 ; test: no br, do next inst
        lhau     prefetch,<offset>(vpc)        ; start prefetch at dest
        b        FetchLoop                     ; go back for next

JSR     addi     temp,prefetch,2               ; adjust return for -2 bias
        add      vpc,vpc,prefetch              ; this is destination
        lha      prefetch,0(vpc)               ; start new prefetch
        stwu     temp,-4(regA7)                ; push return address
        b        FetchLoop                     ; go back for next
```

Branch Prediction

In the best of all possible worlds all the branches in the emulator would be completely predictable and cost no execution cycles. In practice this is very difficult.

A careful reading of the 601 reference manual [1] shows that the best results occur when the instruction execution jumps to continuation code or back to the front of the fetch loop after two or more other instructions, because here it is always an unconditional branch and the integer instructions in front of it completely mask its execution time. Except for the sequence control opcodes (branches, subroutine calls and returns), these jumps can be completely “folded” and effectively execute in zero time, as in the first branch of Figure 2.

Figure 2. Branch Prediction Latency in Emulation

```
  Cycle     1   2   3   4   5   6   7   8   9  10  11  12
(D4AF)                                                              Key:
  lwzx    DC  EX  CW  WB                                              DC   Decode
  lhau      -   DC  EX  CW  WB                                        EX   Execute
  addc.     -   -   DC  EX  WB
  b cont    EX  CW          (branch fully predicted, no delay)
  adde      -   -   -   DC  EX  WB
  b fetch   -   -   EX  CW      (branch fully predicted, no delay)
(fetch)
  rlwimi    -   -   -   -   DC  EX  WB
  mtspr     -   -   -   -   -   DC  EX  WB
  lhau      -   -   -   -   -   -   DC  EX  CW  WB
  bctr      -   -   -   -   =   =   =   EX  CW  (2-cycle delay)
(2F42)
  addc.     -           -           -   -   -   DC  EX  WB
  lhau      -           -           -   -   -   -   DC  EX  CW  WB
  stwx      -           -           -   -   -   -   -   DC  EX
  b fetch   -           -           -   -   -   EX  CW  (no delay)
(fetch)
  rlwimi    -           -           -   -   -   -   -   -   DC  EX  WB
  mtspr     -           -           -   -   -   -   -   -   -   DC  EX  WB
  lhau      -           -           -   -   -   -   -   -   -   -   DC  EX  CW  WB
  bctr      -           -           -   -   -   -   -   -   =   =   =   EX  CW  (2-cycle delay)
(60E0)
  lhau      -           -           -   -   -   -   -   -   -   -   -   -   -   DC  EX  CW  WB
            1           4           7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22
```

CW	*Cache Wait*
WB	*Write Back*
=	*Delayed Execution*

The most serious dent comes from the instruction fetch loop dispatch, where the decode table address jumped to cannot be known until the fetched opcode is inserted into the table address register and that register is transferred to the **ctr** or **lr** register. This means that the first instruction of each table entry is delayed by the time it takes for cache access after the transfer instruction completes, as in the third branch of Figure 2. There is not a lot the emulator can do while waiting for the branch to execute, other than prefetch the next opcode or address word.

One possibility for improvement is to merge the fetch loop into the execution of each opcode, so that instruction decoding effectively overlaps the execution of the previous opcode. This makes the emulator much larger and more complex, but it can also improve performance substantially, as shown in Figure 3. This is possible because (except again in the case of sequence control operations) the location of the next opcode can be completely known as soon as the current opcode is decoded. Note that each opcode routine still includes a prefetch, but now that is the operand address of the next instruction (or the following opcode, in the case of 2-byte instructions). The result is that the third 68K opcode of the example begins to execute three clocks sooner than in Figure 2, for a net improvement of about 20%. The best performance, if there is nothing else to do at all (for example, NOP), is seven clocks from opcode decode to opcode decode, which leaves four of the seven clocks for emulation execution. This is the execution rate in Figure 3. Figure 3 also assumes a 32-byte table entry size, but it is probably unnecessary for best performance most of the time. For both examples we have assumed perfect cache hits for all opcode fetches and branches.

Even the most careful reading of the 601 reference manual will fail to disclose yet another nasty performance hit that attacks distributed programs such as emulators. This comes from the instruction prefetch logic. The instruction queue in the 601 is eight instructions (32 bytes), and as soon as there is a vacancy in the queue, the prefetch logic begins to ask the cache for the next 32 bytes. The manual tells us it will only go to main memory when the queue is half empty, but (and the manual does not say this) it can tie up the cache for one or more cycles before that, even if there is a fully predictable branch in the queue. The result is that the optimum rate of seven clocks per 68K instruction is achievable only when 32-byte table entries are executed in consecutive order, which is of course not possible in an emulator. You might think that the **bctr** instruction would execute in the same time regardless of the location of the target address (so long as it's in the cache), but it turns out that when it requests the same cache line as the in-process prefetch, there is no penalty; otherwise it must wait for the next available cache cycle. There appears to be no defense against this penalty: it's a cost of doing business in the 601.

Figure 3. Improved Branch Prediction by Interleaved Fetch/Execute

```
  Cycle     1   2   3   4   5   6   7   8   9   10  11  12
(D4AF)
```

Key:

```
  lhau      DC  EX  CW  WB
  lwzx      -   DC  EX  CW  WB
  rlwimi    -   -   DC  EX  WB
  mtspr     -   -   -   DC  EX  WB
  addc.     -   -   -   -   DC  EX  WB
  lhau      -   -   -   -   -   DC  EX  CW  WB
  adde      -   -   -   -   -   -   DC  EX  WB
  bctr      -   -   -   -   =   EX  CW        (branch predicted, no delay)
(2F42)
  lhau      -           -           -   DC  EX  CW  WB
  addc.     -           -           -   -   DC  EX  WB
  rlwimi    -           -           -   -   -   DC  EX  WB
  mtspr     -           -           -   -   -   -   DC  EX  WB
  stwx      -           -           -   -   -   -   -   DC  EX
  lhau      -           -           -   -   -   -   -   -   DC  EX  CW  WB
  bctr      -           -           -   -   -   -   =   =   EX  CW   (1-cycle delay)
(60E0)
  lhau      -           -           -   -   -   -   -   -   -   -   DC  EX  CW  WB
            1           4           7   8   9   10  11  12  13  14  15  16  17  18  19  20
```

DC	*Decode*
EX	*Execute*
CW	*Cache Wait*
WB	*Write Back*
=	*Delayed Execution*

Other Issues

One of the free benefits of table-lookup emulation is that the simpler A-trap system calls can be directly emulated. Instead of going through the A-trap dispatcher, traps like SetPt, PtInRect, BitAnd, and the like can be directly emulated as if they were 68000 instructions. The result could speed up these routines as much as ten times. Probably for compatibility with trap-patching, Apple chose not to do this, but it has some interesting possibilities. Whoever patches these utilities, anyway?

David Ramsey reported in *MacWeek* that about 25K of the 601 on-chip cache is filled with the Apple emulator; much of this would be unused table entries. Consider that each cache sector is 32 bytes, so a single 8-byte table entry will drag three of its neighbors into the cache with it. In most cases the commonly-used opcodes have infrequently-used neighboring opcodes: for example, register-offset addressing is common for A6 (the local stack frame) but much less common for A4 and A5, and almost never for A7; yet all four decode table entries are part of the same 32-byte block and come into the cache together. Odd-address short branches are illegal, but two of them fill space in the cache each time a valid even-address branch is executed. Using a 16-byte table entry still brings in some unused code (50% instead of 75%), but with three times more available code space right in the table entry, there is much less overflow code to load in. A 32-byte table entry loads only the executing code, with almost no overflow, except that the prefetch look-ahead will tend to get another 32 bytes anyway. However, only in the overflow area is there a possibility of code shared between several opcodes, but then only if the table entry is large enough to fully accomodate the decode details. The illegal instructions (which are never executed in a correct program) would waste most of their table space, but this can be largely recovered by redirecting it to continuations of other (longer) opcode sequences. For example, the JSR opcode above requires five instructions; three fill its place in a 16-byte table entry, and the remaining two can be dropped into any 8-byte hole elsewhere in the table. Depending on the relative frequency of such longer instructions, they could even be split across multiple table holes at no execution cost. The Apple development team undoubtedly considered all of this in the light of the increased ROM cost of a larger table size.

Performance Results

Here is a little program fragment I used to test the interpreter performance. It calculates the square root of 10,000 by adding successive odd integers. The inner loop executes 1,000,000 times. On the IIci (a plain 25MHz 68030 with no cache) it executes in 106 ticks (see Table 3); on the 6100 using Apple's emulator it takes 79 ticks. My emulator described here, using a 16-byte table entry without interleaving, runs in 113 ticks; with a 32-byte interleaved table entry that comes down to 93 ticks. The roughly equivalent native PowerPC code for the same algorithm ran in 10 ticks; optimized to use registers instead of memory for local variables, it ran in a little over three ticks. The register-optimized 601 code has a calculated inner loop transit time of three clocks, compared to 11 clocks in the unoptimized version; this implies in both cases an execution rate of 1,000,000 clocks per tick, which is almost exactly 60Mhz, the rated CPU speed. The calculated minimum inner-loop transit time in my emulator is 58 clocks.

Table 3. Relative Performance of 10,000 Integer Square Roots

Platform	Ticks	Clocks
plain IIci	106	
6100 emulator	79	82
my 16-byte table	113	
my 32-byte table	93	70
Apple’s 8-byte table	127	
native 60MHz 601	10	
optimized 601	3.3	

It’s hard to know whether the degradation in actual performance as compared to the theoretical is due to interrupts running under Apple’s emulator flushing my interpreter out of the cache or address translation buffer, or some other cause. A copy of Apple’s emulator running in my test bed ran the same program in 127 ticks.

In the “Good Old Days” we dealt with a problem of this sort by connecting a logic analyzer to the system and tracing the bus transactions. The PowerPC cache is so big that once it has loaded up the running program and its data (especially a tiny program like this), there are no bus transactions to trace. It takes a whole new way of thinking to devise tests for getting past this kind of barrier. Get past it I eventually did, and I was able to precisely measure the inner loop transit time, which is how I discovered the statistical 1.5 wasted fetch-ahead cycles in both Apple’s and my emulators. The third column in Table 3 gives actual loop transit times for my best emulator and the Apple ROM, measured in processor clocks.

Another problem with analyzing the emulator performance is that the Apple emulator is accessible only by dropping into supervisor mode, while everything else (except low-level interrupt routines) runs in user mode. This includes debuggers, so they cannot step through the emulator. Furthermore, MacsBug has so grown in complexity and dependence on the system software that it no longer robustly commands full control of the computer; there does not even exist a 601 equivalent to the original MacsBug in its glory days. That makes debugging mixed-mode software particularly difficult, and the Apple PowerPC team has been remarkably reluctant to help. About the only way to see what is actually happening in the machine is to turn off caching for the emulator, then watch the bus transactions on a logic analyzer. On the other hand, this paper might not have been possible if critical information were available only from them under non-disclosure.

Listing 4 is the HyperTalk source code for the test routine and Listing 5 is its compiled 68000 object code disassembled (with the inner loop marked); Listing 6 is corresponding the register-optimized PowerPC code.

Listing 4. Test Program Code (HyperTalk Source)

```
put 1 into ntimes
repeat until ntimes=10000
  add 1 to ntimes
  put 1 into oddly
  put 1 into dvdnd
  repeat until dvdnd=10000
    add 2 to oddly
    put oddly+dvdnd into dvdnd
    end repeat
  end repeat
```

Listing 5. Test Program Code (Disassembled 68000)

```
+002E  MOVEQ      #$01,D1              | 7201
+0030  MOVE.L     D1,$000C(A7)         | 2F41 000C
+0034  MOVE.L     #$00002710,D3        | 263C 0000 2710
+003A  CMP.L      $000C(A7),D3         | B6AF 000C
+003E  BEQ        test+$0074           | 6700 0034
+0042  ADDQ.L     #$1,$000C(A7)        | 52AF 000C
+0046  MOVEQ      #$01,D1              | 7201
+0048  MOVE.L     D1,$0008(A7)         | 2F41 0008
+004C  MOVEQ      #$01,D1              | 7201
+004E  MOVE.L     D1,$0004(A7)         | 2F41 0004
+0052  MOVE.L     #$00002710,D3        | 263C 0000 2710 +
+0058  CMP.L      $0004(A7),D3         | B6AF 0004      |
+005C  BEQ        test+$0072           | 6700 0014      |
+0060  ADDQ.L     #$2,$0008(A7)        | 54AF 0008      |
+0064  MOVE.L     $0004(A7),D2         | 242F 0004      |
+0068  ADD.L      $0008(A7),D2         | D4AF 0008      |
+006C  MOVE.L     D2,$0004(A7)         | 2F42 0004      |
+0070  BRA.S      test+$0052           | 60E0           +
+0072  BRA.S      test+$0034           | 60C0
```

Listing 6. Test Program Code (Partly-Optimized 601)

```
0000014   3B80 0001                    li          28,1
                                 outerloop
0000018   2C1C 2710                    cmpi   0,0,28,10000
000001C   4182 0028                    beq         stop
0000020   3B9C 0001                    addi   28,28,1
0000024   3B00 0001                    li          24,1
0000028   3A80 0001                    li          20,1
                                 sqrtloop
000002C   2C14 2710                    cmpi   0,0,20,10000
0000030   4182 0010                    beq         done
0000034   3B18 0002                    addi   24,24,2
0000038   7E94 C214                    add         20,20,24
000003C   4BFF FFF0                    b           sqrtloop
0000040   4BFF FFD8              done  b      outerloop
```

Future Directions

After this paper went to press, *MacWeek* [5] reported a new developmental Apple emulator that "runs at twice the speed of its predecessor." A pure emulator to do this is simply not possible. It is possible, however, to get this kind of performance out of a simple on-the-fly code-translation scheme, provided that the translated code is cached for repeated execution.

A simple translator can be derived from the emulator by separating the fetch and decode loop from the emulation proper, and replacing the emulation part with something to copy those instructions into the translated code cache. The instructions to copy can be essentially identical to those used for emulation, except that inline constants would replace register references where appropriate. The cost of translation would be somewhat greater than the cost of emulation, but that is more than compensated by the fact that it is not repeated during multiple loop executions. Using this technique on the inner loop of Listing 5 without any optimization at all collapses it from 70 clocks to 20, as shown in Listing 7. Adding simple instruction scheduling to eliminate cache-hit latencies, and peephole optimization to eliminate redundant CC operations and register loads reduces it down to the 11 clocks of the unoptimized compiler output. These two optimizations are fairly easy to accomplish without a major performance burden (for algorithms, see Pittman&Peters [4]), but further optimizations become excessively expensive for diminishing returns.

Listing 7. Test Program Code Inner Loop, Translated to 601

```
       loop addic.  rD3,0,#10000           ; MOVE.L: also set CC
           lzw     temp,4(rA7)             ; CMP.L: fetch long data
```

```
        subfc.  temp,temp,rD3           ;   throw result away, keep CC
        subfe   temp,0,0                ;   invert the carry
        addc    temp,temp,temp          ;   copy it back to C in XER
        bc      12,2,exit               ; BEQ
        lzw     temp,8(rA7)             ; ADDQ.L: fetch long data
        addic.  temp,temp,#2            ;   add & set CC
        adde    Xreg,0,0                ;   copy out X bit
        stx     temp,8(rA7)             ;   store it back
        lzw     rD2,4(rA7)              ; MOVE.L: fetch long data
        addic.  rD2,rD2,#0              ;   set CC
        lzw     temp,8(rA7)             ; ADD.L: fetch long data
        addc.   rD2,rD2,temp            ;   add & set CC
        adde    Xreg,0,0                ;   copy out X bit
        stw     rD2,4(rA7)              ; MOVE.L: store long data
        addic.  rD2,rD2,#0              ;   also set CC
        b       loop   ; BRA.S
    exit ...
```

The allocation and aging of memory for caching the translated code comes with its own special set of problems. Any time the VM modifies code, the translated 601 code must be discarded and retranslated. An easy way to detect this would be to set the write protection of the original 68K code whenever it is translated, then clear the protection and flush the translated code at the same time when the protection is first violated. Although self-modifying code is considered poor form, the fact is that every time a code resource is loaded or relocated, it will probably fall out of the translation cache. Clever memory-management hacks could minimize the relocation of unlocked but translated code resources. These are the sorts of things that lower the performance advantage from our first-blush 6x back down to something closer to the reported 2x.

Disclaimer

The information in this paper is based entirely on the author's knowledge and experience in developing virtual machine interpreters in other environments [4], augmented by the use of tools and devices available without restriction on the open market. It contains no Apple secrets (they never gave me any). Most of this paper and the emulator it describes were written while I was still unable to locate and examine the Apple code.

References

[1] *M68000 Family Programmer's Reference Manual*, Motorola M68000PM/AD
[2] *PowerPC 601 User's Manual*, Motorola MPC601UM/AD
[3] *Inside Macintosh: PowerPC System Software* (Addison-Wesley)
[4] Pittman&Peters, *The Art of Compiler Design*, ch.8,9 (Prentice Hall)
[5] *MacWeek*, 06.13.94, p.1.