

Flight Virtual Machines And Metacompilation

Charles Eric LaForest

April 2006
University of Waterloo
Independent Studies

Abstract

A number of Gullwing virtual machines are written in Flight and their relative overheads are compared. The sources of the overhead are explained in terms of the Popek and Goldberg requirements for machine virtualization, which are then used to derive the necessary changes to the Gullwing architecture to optimally support virtual machines. Some minor alterations are performed on the Flight language kernel to allow for retargeting its compilation. From this is built a framework for expressing the Flight kernel in Flight instead of assembly. When combined with the virtual machines, this allows Flight to create a second, nested instance of itself which cannot affect the first.

Contents

1	Virtual Machines	3
1.1	Preambles	3
1.1.1	Opcodes	3
1.1.2	VM Parameters	3
1.1.3	Instruction Call Table	4
1.2	VM1	4
1.3	VM2	6
1.3.1	VM2-1 (Tail-Call Elimination)	7
1.4	VM3	8
1.4.1	Stacks	8
1.4.2	Temporary Storage Congestion	8
1.4.3	Trampoline	8
1.4.4	Source	8
1.4.5	VM3-1 (Trampoline Removed)	10
1.5	VM4	12
1.5.1	VM4-1 (Trampoline Removed)	15
1.6	Execution Overhead Comparison	16
1.6.1	Memory Protection	16
1.6.2	Tail-Call Elimination and Inlining	16
1.6.3	Virtual Stacks vs. Physical Stacks	17
1.6.4	Location of PC and ISR	17
1.6.5	Instruction Extraction	17
2	Virtualization Requirements	18
2.1	Virtual Machine Properties	18
2.2	Instruction Set Architecture Properties	18
2.3	Theorem	18
2.4	Making Gullwing Virtualizable	18
2.5	A Hypothetical VM5	19
2.5.1	Final Improvement	19
2.5.2	Source Code Draft	19
3	Metacompiler	21
3.1	Limitations	21
3.2	Source	21
3.3	Flight In Flight	23
A	Flight Kernel Changes (v. 6)	26
B	Flight Extension Changes	26
	References	27

1 Virtual Machines

The virtual machines (VMs) presented are behavioural simulations of the Gullwing processor [LaF05a]¹. It is important to keep in mind that none of these VMs do address translation². The addresses used are the physical ones at which the allocated memory resides. This means that the VMs are imperfect in their simulation, since software could test for the location of the Kernel and Input/Output ports to see if they match those on the native hardware (which they wouldn't).

1.1 Preambles

These are blocks of code referenced throughout. Where they would be inserted in-line, they are replaced for brevity by a reference between angle brackets (<...>). All code listed makes use of the Flight Extensions, described in [LaF05b].

1.1.1 Opcodes

These words return the opcode for each instruction.

```
: PCFETCHopcode      n# 0 ;
: CALLopcode         n# 1 ;
: RETopcode          n# 2 ;
: JMPopcode          n# 3 ;
: JMPZEROopcode      n# 4 ;
: JMPPLUSopcode      n# 5 ;
: FETCHAopcode       n# 6 ;
: STOREAopcode       n# 7 ;
: FETCHAPLUSopcode   n# 8 ;
: STOREAPLUSopcode   n# 9 ;
: FETCHRPLUSopcode   n# 10 ;
: STORERPLUSopcode   n# 11 ;
: LITopcode          n# 12 ;
: UNDEF0opcode       n# 13 ;
: UNDEF1opcode       n# 14 ;
: UNDEF2opcode       n# 15 ;
: XORopcode          n# 16 ;
: ANDopcode          n# 17 ;
: NOTopcode          n# 18 ;
: TWOSTARopcode      n# 19 ;
: TWOSLASHopcode     n# 20 ;
: PLUSopcode         n# 21 ;
: PLUSSTARopcode     n# 22 ;
: DUPopcode          n# 23 ;
: DROPopcode         n# 24 ;
: OVERopcode         n# 25 ;
: TORopcode          n# 26 ;
: RFROMopcode        n# 27 ;
: TOAopcode          n# 28 ;
: AFROMopcode        n# 29 ;
: NOPopcode          n# 30 ;
: UNDEF3opcode       n# 31 ;
```

1.1.2 VM Parameters

These are common to all the virtual machines (VM). They define the size of the memory and of the stacks, the width of the opcodes, and calculate the locations of the virtual memory-mapped input and output ports, which are always the last two locations in MEM.

```
: MEMSIZE            n#      8192 ;
: MEM                MEMSIZE var ;
: STACKDEPTH         n#      16 ;
: OPCODEWIDTH        n#       5 ;
: OPCODEMASK         n#     31 ;
: MEM_INPUT          MEM MEMSIZE + -n 2 + # ;
: MEM_OUTPUT         MEM MEMSIZE + -n 1 + # ;
```

¹The state machine description in that report has a few errors and omissions, which are corrected in [LaF05d].

²There was not enough time to implement that feature.

1.1.3 Instruction Call Table

This creates an array of the addresses of the VM functions which emulate the native instructions. The `&>` function looks up the address and stores it into memory. Calling `instruction_call_table` returns the address of the array, which is then indexed into with the opcode to be emulated.

```

: , c #>c c ALIGN ;
: &> c l c , ;

: instruction_call_table
create
&> do_pcfetch
&> do_call
&> do_ret
&> do_jump
&> do_jumpzero
&> do_jumpplus
&> do_fetcha
&> do_storea
&> do_fetchaplus
&> do_storeaplus
&> do_fetchrplus

```

```

&> do_storerplus
&> do_lit
&> do_undef
&> do_undef
&> do_undef
&> do_xor
&> do_and
&> do_not
&> do_twostar
&> do_twoslash
&> do_plus
&> do_plusstar
&> do_dup
&> do_drop
&> do_over
&> do_tor
&> do_rfrom
&> do_toa
&> do_afrom
&> do_nop
&> do_undef
does ;

```

1.2 VM1

The first virtual machine is an almost pure emulation of the Gullwing processor. The stacks are reproduced in main memory, as is the A register (AREG), but the Program Counter (PC) and the Instruction Shift Register (ISR) are kept on the Data Stack, with the ISR on top.

```

<VM Parameters (Section 1.1.2)>

: AREG n 1 var ;

```

These define the Data and Return Stacks (DSH, RSH), compute the address of their tails (DST, RST), and create pointers into them (DSP, RSP).

```

: DSH STACKDEPTH var ;
: DST DSH STACKDEPTH -n 1 + + # ;
: DSP n 1 var ;
: RSH STACKDEPTH var ;
: RST RSH STACKDEPTH -n 1 + + # ;
: RSP n 1 var ;

```

Set the pointers to the address of the heads of the stacks.

```

: empty_stacks
DSH # DSP # (!) RSH # RSP # (!) ;

```

The following block constructs the push and pop operations for the stacks. The pointer incrementing and decrementing functions only do so if the pointer is not already at either the tail or the head, to prevent stack overflows and underflows from corrupting other memory.

```

: DSP1+
DSP # (@) (DUP) DST # (XOR)

```

```

if (N+) 1 (DUP) (A!) ; else ;
: DSP1-
DSP # (@) (DUP) DSH # (XOR)
if (DUP) (N-) 1 (A!) ; else ;
: DSPprev
DSP # (@) (DUP) DSH # (XOR)
if (N-) 1 ; else ;
: RSP1+
RSP # (@) (DUP) RST # (XOR)
if (N+) 1 (DUP) (A!) ; else ;
: RSP1-
RSP # (@) (DUP) RSH # (XOR)
if (DUP) (N-) 1 (A!) ; else ;
: push_ds c DSP1+ (!) ;
: pop_ds c DSP1- (@) ;
: push_rs c RSP1+ (!) ;
: pop_rs c RSP1- (@) ;

```

These redirect memory reads and writes to the actual I/O ports if the emulated ones are accessed, else they simply read or write from memory. The addresses used are untranslated, and there is no protection against accessing memory outside of MEM.

```

: mem_read
(DUP) MEM_INPUT # (XOR)
if (@) ; else (DROP) c ># ;
: mem_write (DUP) MEM_OUTPUT # (XOR)

```

```
if (!) ; else (DROP) c #> ;
```

The following functions emulate the Gullwing instructions, which are described in [LaF05a] and summarized in [LaF05b]. They typically fetch from the stacks in memory, perform the work on the actual Data Stack, then store the results back.

For example, `do_call` begins by dropping the ISR from the stack, duplicates the PC, increments that copy by 1 to make it into the return address, and stores it on the emulated return stack. It then uses the original PC to fetch the called address, makes a copy and puts it away into the A register, increments the original by 1 to make it into the next address to execute, and finally uses the copy to fetch the new block of instructions, which will be processed by `do_next_instruction` later on.

```
: do_call
(DROP) (DUP) (N+) 1 c push_rs
(@) (DUP) (>A) (N+) 1 (A@) ;

: do_pcfetch
(DROP) (DUP) (>A) (N+) 1 (A@) ;

: do_ret
(DROP) (DROP) c pop_rs
(DUP) (>A) (N+) 1 (A@) ;

: do_jump
(DROP) (@) (DUP) (>A) (N+) 1 (A@) ;

: do_jumpzero
c pop_ds
if (>R) (N+) 1 (R>) ;
else j do_jump

: do_jumpplus
c pop_ds
if- (>R) (N+) 1 (R>) ;
else j do_jump

: do_fetcha
AREG # (@) c mem_read c push_ds ;

: do_storea
c pop_ds AREG # (@) c mem_write ;

: do_fetchaplus
AREG # (@) (DUP) c mem_read c push_ds
(N+) 1 AREG # (!) ;

: do_storeaplus
AREG # (@) c pop_ds (OVER) c mem_write
(N+) 1 AREG # (!) ;

: do_fetchrplus
c pop_rs (DUP) c mem_read c push_ds
(N+) 1 c push_rs ;
```

```
: do_storerplus
c pop_rs c pop_ds (OVER) c mem_write
(N+) 1 c push_rs ;

: do_lit
(>R) (DUP) (>A) (N+) 1 (R>) (A@) c push_ds ;

: do_undef
n 31 COMPILE_OPCODE ;

: do_xor
c pop_ds DSP # (@) (@) (XOR) (A!) ;

: do_and
c pop_ds DSP # (@) (@) (AND) (A!) ;

: do_not
DSP # (@) (@) (NOT) (A!) ;

: do_twostar
DSP # (@) (@) (2*) (A!) ;

: do_twoslash
DSP # (@) (@) (2/) (A!) ;

: do_plus
c pop_ds DSP # (@) (@) (+) (A!) ;

: do_plusstar
c pop_ds (>R) DSP # (@) (@)
(R>) (+*) c push_ds (DROP) ;

: do_dup
DSP # (@) (@) c push_ds ;

: do_drop
c pop_ds (DROP) ;

: do_over
c DSPprev (@) c push_ds ;

: do_tor
c pop_ds c push_rs ;

: do_rfrom
c pop_rs c push_ds ;

: do_toa
c pop_ds AREG # (!) ;

: do_afrom
AREG # (@) c push_ds ;

: do_nop
(NOP) ;

<Instruction Call Table (Section 1.1.3)>
```

This compiles code³ which shifts out the previous instruction, shifting in zeroes which eventually form the PC@ opcode.

```
: (shift_isr)
c (2/) c (2/) c (2/) c (2/) c (2/) ;
```

This compiles code to mask off a copy of the ISR, returning the current instruction to emulate.

```
: (extract_instruction)
c (DUP) OPCODEMASK # c # c (AND) ;
```

The extracted instruction is then used to index into the instruction call table to get the address of the emulating function, which is then called.

```
: do_next_instruction
```

1.3 VM2

The second virtual machine is identical to the first, but memory accesses and program flow changes are now checked to see if they fall within the bounds of MEM. Only code added or changed is listed below.

```
: MEMHEAD MEM # ;
: MEMTAIL MEM_OUTPUT # ;
```

If the address is after the head of MEM, then (check_low) will return a positive number.

```
: (check_low)
MEMHEAD negate # c # c (+) ;
```

If the address is before the tail of MEM, then (check_high) will return a positive number.

```
: (check_high)
c (negate) MEMTAIL # c # c (+) ;
```

Perform both checks and OR the numbers together. If either is negative, indicating a memory access outside of MEM, then the final number will be necessarily also negative.

```
: mem_in_range?
(DUP) (check_low) (OVER) (check_high) (OR)

: mem_access_msg
create >$ ILLEGAL_MEMORY_ACCESS: $>c
does j cs>
```

Outputs the error message, followed by the address in question.

```
: report_mem_error
c mem_access_msg
c \s c #> j \n
```

If the address is out of range, then output the error message, and echo the remaining input to the output as a crude way of indicating where the range violation occurred.

```
(extract_instruction)
(>R) (shift_isr) (R>)
instruction_call_table # (+) (@)
c EXECUTE ;
```

Finally, the VM is started by emptying the stacks, placing initial values of PC and ISR on the Data Stack, and entering an endless loop which emulates the instructions.

```
: run_vm
c empty_stacks
MEM #
n 0 #
: vm_loop
c do_next_instruction
j vm_loop
```

```
: access_check
c mem_in_range?
if-
c report_mem_error
j errcontext
else ;
```

The access_check function is then inserted just before any emulated memory access code would use an address. Flow control instructions must be included since self-modifying code could pass a compile-time memory range check, but generate illegal addresses at run-time.

```
: mem_read
(DUP) MEM_INPUT # (XOR)
if
c access_check (@) ;
else
(DROP) c ># ;

: mem_write
(DUP) MEM_OUTPUT # (XOR)
if
c access_check (!) ;
else
(DROP) c #> ;
```

```
: do_pcfetch
(DROP) c access_check
(DUP) (>A) (N+) 1 (A@) ;
```

```
: do_call
(DROP) (DUP) (N+) 1 c push_rs (@)
c access_check (DUP) (>A) (N+) 1 (A@) ;
```

³It is a naming convention in Flight to place inside parentheses the names of functions which compile their code instead of executing it.

```

: do_ret
(DROP) (DROP) c pop_rs c access_check
(DUP) (>A) (N+) 1 (A@) ;

: do_jump
(DROP) (@) c access_check
(DUP) (>A) (N+) 1 (A@) ;

```

1.3.1 VM2-1 (Tail-Call Elimination)

To eliminate some of the function call overhead, function tail-calls have been replaced by jumps, and the call to EXECUTE in do_next_instruction has been replaced with in-line code. These changes are listed below. The improvement in performance is discussed in Section 1.6.

```

: (check_low)
MEMHEAD negate # c # j (+)

: (check_high)
c (negate) MEMTAIL # c # j (+)

: mem_read
(DUP) MEM_INPUT # (XOR)
if
c access_check (@) ;
else
(DROP) j >#

: mem_write
(DUP) MEM_OUTPUT # (XOR)
if
c access_check (!) ;
else
(DROP) j #>

: do_fetcha
AREG # (@) c mem_read j push_ds

: do_storea
c pop_ds AREG # (@) j mem_write

: do_fetchrplus
c pop_rs (DUP) c mem_read
c push_ds (N+) 1 j push_rs

: do_storerplus
c pop_rs c pop_ds (OVER)
c mem_write (N+) 1 j push_rs

: do_lit
(>R) (DUP) (>A) (N+) 1
(R>) (A@) j push_ds

: do_dup
DSP # (@) (@) j push_ds

: do_over
c DSPprev (@) j push_ds

: do_tor
c pop_ds j push_rs

: do_rfrom
c pop_rs j push_ds

: do_afrom
AREG # (@) j push_ds

: do_next_instruction
(extract_instruction)
(>R) (shift_isr) (R>)
instruction_call_table # (+) (@)
(>R) ;

```

1.4 VM3

1.4.1 Stacks

Virtual stacks kept in memory require many memory accesses per emulated instruction. The third virtual machine eliminates them and instead uses the actual Data and Return stacks as they would normally be used. The PC and ISR are still kept on the Data Stack and are moved out of the way as needed. The performance impact is discussed in Section 1.6.

1.4.2 Temporary Storage Congestion

Since there are only three non-memory places to temporarily store information (A Register, Return Stack, Data Stack), keeping more than two items (PC and ISR) would cause a great increase in the amount of shuffling required to get to the information buried underneath on the stacks. When `do_next_instruction` calls an instruction emulation function, it would place a return address on the Return Stack, which raises the total of saved items to three, and thus greatly complicates emulation.

The solution used here is to change the call/return sequence into a pair of jumps. The first is in `do_next_instruction`, which jumps to `EXECUTE` instead of calling it, which effectively jumps to the provided function address. The second is by exiting functions with a jump back to `do_next_instruction`, instead of a tail-call, tail jump, or function return. Together, these avoid the use of the Return Stack.

1.4.3 Trampoline

Unfortunately, a set of dependencies prevents the jump back to `do_next_function` from being straightforward. The `instruction_call_table` can only be compiled after the functions it indexes are, and `do_next_function` can only be compiled after the `instruction_call_table` is. This means that the emulation functions cannot know the address of `do_next_instruction` since it has not been compiled yet!

The solution is to replace the jump to `do_next_instruction` with code for an indirect jump which takes its address from a previously defined variable. Just before the VM is started, this address is set to that of `do_next_instruction`. This trampoline is however much less efficient than the call/return pair it replaces.

It should be possible to use actual jumps whose targets would then be fixed-up afterwards, as is done in the Flight Extensions for the `if`, `else`, `create`, and `does` constructs, but doing so for an arbitrary number of functions is not trivial, since the number of addresses to fix-up is not known in advance and may exceed the capacity of the Data Stack.

1.4.4 Source

```
: trampoline n 1 var ;                               c mem_access_msg
                                                       c \s c #> j \n

: (jump_to_trampoline)
trampoline # c # c (@) c (>R) c ; ;                   : access_check
                                                       c mem_in_range?
<VM Parameters (Section 1.1.2)>                       if-
                                                       c report_mem_error
                                                       j errcontext
                                                       else ;

: (check_low)                                         : AREG n 1 var ;
MEMHEAD negate # c # c (+) ;

: (check_high)                                       : mem_read
c (negate) MEMTAIL # c # c (+) ;                     (DUP) MEM_INPUT # (XOR)
                                                       if
                                                       c access_check (@) ;
                                                       else

: mem_in_range?                                       : mem_write
(DUP) (check_low) (OVER) (check_high) (OR) ;(DROP) c ># ;
                                                       (DUP) MEM_OUTPUT # (XOR)
                                                       if
                                                       c access_check (!) ;
                                                       else

: mem_access_msg                                     : mem_write
create >$ ILLEGAL_MEMORY_ACCESS: $>c                 (DUP) MEM_OUTPUT # (XOR)
does j cs>                                           if
                                                       c access_check (!) ;
                                                       else

: report_mem_error
```

```

(DROP) c #> ;

: do_pcfetch
(DROP) c access_check
(DUP) (>A) (N+) 1 (A@)
(jump_to_trampoline)

: do_call
(DROP) (DUP) (N+) 1 (>R) (@)
c access_check
(DUP) (>A) (N+) 1 (A@)
(jump_to_trampoline)

: do_ret
(DROP) (DROP) (R>)
c access_check
(DUP) (>A) (N+) 1 (A@)
(jump_to_trampoline)

: do_jump
(DROP) (@)
c access_check
(DUP) (>A) (N+) 1 (A@)
(jump_to_trampoline)

: do_jumpzero
(>R) (>R)
if
(R>) (N+) 1 (R>)
(jump_to_trampoline)
else
(R>) (R>)
j do_jump

: do_jumpplus
(>R) (>R)
if-
(R>) (N+) 1 (R>)
(jump_to_trampoline)
else
(R>) (R>)
j do_jump

: do_fetcha
(>R) (>R)
AREG # (@) c mem_read
(R>) (R>)
(jump_to_trampoline)

: do_storea
(>R) (>R)
AREG # (@) c mem_write
(R>) (R>)
(jump_to_trampoline)

: do_fetchaplus
(>R) (>R)

AREG # (@) (DUP)
c mem_read (>R)
(N+) 1 AREG # (!) (R>)
(R>) (R>)
(jump_to_trampoline)

: do_storeaplus
(>R) (>R)
(>R) AREG # (@) (R>) (OVER)
c mem_write
(N+) 1 AREG # (!)
(R>) (R>)
(jump_to_trampoline)

: do_fetchrplus
(R>) (DUP)
c mem_read (>A) (N+) 1 (>R)
(>R) (>R)
(A>)
(R>) (R>)
(jump_to_trampoline)

: do_storerplus
(R>) (>A)
(>R) (>R)
(>R) (A>) (R>)
(OVER) c mem_write (N+) 1 (>A)
(R>) (R>)
(A>) (>R)
(jump_to_trampoline)

: do_lit
(>R) (DUP) (>A) (N+) 1 (>R) (A@)
(R>) (R>)
(jump_to_trampoline)

: do_undef
n 31 COMPILER_OPCODE
(jump_to_trampoline)

: do_xor
(>R) (>R)
(XOR)
(R>) (R>)
(jump_to_trampoline)

: do_and
(>R) (>R)
(AND)
(R>) (R>)
(jump_to_trampoline)

: do_not
(>R) (>R)
(NOT)
(R>) (R>)
(jump_to_trampoline)

```

```

: do_twostar
(>R) (>R)
(2*)
(R>) (R>)
(jump_to_trampoline)

: do_twoslash
(>R) (>R)
(2/)
(R>) (R>)
(jump_to_trampoline)

: do_plus
(>R) (>R)
(+)
(R>) (R>)
(jump_to_trampoline)

: do_plusstar
(>R) (>R)
(++*)
(R>) (R>)
(jump_to_trampoline)

: do_dup
(>R) (>R)
(DUP)
(R>) (R>)
(jump_to_trampoline)

: do_drop
(>R) (>R)
(DROP)
(R>) (R>)
(jump_to_trampoline)

: do_over
(>R) (>R)
(OVER)
(R>) (R>)
(jump_to_trampoline)

: do_tor
(>A) (OVER) (>R)
(>R) (DROP) (R>) (A>)

(jump_to_trampoline)

: do_rfrom
(R>) (>A)
(>R) (>R)
(A>)
(R>) (R>)
(jump_to_trampoline)

: do_toa
(>R) (>R)
AREG # (!)
(R>) (R>)
(jump_to_trampoline)

: do_afrom
(>R) (>R)
AREG # (@)
(R>) (R>)
(jump_to_trampoline)

: do_nop
(NOP)
(jump_to_trampoline)

<Instruction Call Table (Section 1.1.3)>

: (shift_isr)
c (2/) c (2/) c (2/) c (2/) c (2/) ;

: (extract_instruction)
c (DUP) OPCODEMASK # c # c (AND) ;

: do_next_instruction
(extract_instruction)
(>R) (shift_isr) (R>)
instruction_call_table # (+) (@)
j EXECUTE

: run_vm
MEM #
n 0 #
l# do_next_instruction
trampoline # (!)
j do_next_instruction

```

1.4.5 VM3-1 (Trampoline Removed)

The trampoline mechanism was removed and tail-call elimination and inlining were added as in Section 1.3.1. As expected, removing the trampoline forces some very convoluted stack shuffling to take place, especially in functions that emulate instructions that use both stacks such as `do_fetchrplus`, `do_storerplus`, `do_rfrom`, and `do_tor`. Only the code significantly affected by these changes is listed.

```

: mem_read
(DUP) MEM_INPUT # (XOR)
if
c access_check (@) ;
else
(DROP) j >#

```

```

: mem_write
(DUP) MEM_OUTPUT # (XOR)
if
  c access_check (!) ;
else
  (DROP) j #>

: do_call
(R>) (>A)
(DROP) (DUP) (N+) 1 (>R)
(A>) (>R)
(@) c access_check
(DUP) (>A) (N+) 1 (A@) ;

: do_ret
(R>) (>A)
(DROP) (DROP) (R>)
(A>) (>R)
c access_check
(DUP) (>A) (N+) 1 (A@) ;

: do_jmpzero
(>R) (>R)
if
  (R>) (N+) 1 (R>) ;
else
  (R>) (R>)
  j do_jump

: do_jmpplus
(>R) (>R)
if-
  (R>) (N+) 1 (R>) ;
else
  (R>) (R>)
  j do_jump

: do_fetcha
(>R) (>R)
AREG # (@) c mem_read
(R>) (R>) ;

: do_storea
(>R) (>R)
AREG # (@) c mem_write
(R>) (R>) ;

: do_fetchaplus
(>R) (>R)
AREG # (@) (DUP)
c mem_read (>R)
(N+) 1 AREG # (!) (R>)
(R>) (R>) ;

: do_storeaplus
(>R) (>R)
(>R) AREG # (@) (R>) (OVER)
c mem_write
(N+) 1 AREG # (!)
(R>) (R>) ;

: do_fetchrplus
(R>)
(R>) (DUP)
c mem_read
(>A) (N+) 1 (>R)
(>R)
(>R) (>R)
(A>)
(R>) (R>) ;

: do_storerplus
(R>)
(R>) (>A)
(>R) (>R) (>R)
(>R) (A>) (R>)
(OVER) c mem_write
(N+) 1 (>A)
(R>) (R>) (R>)
(A>) (>R)
(>R) ;

: do_tor
(>R) (>R)
(>A)
(R>) (R>) (R>)
(A>) (>R)
(>R) ;

: do_rfrom
(R>)
(R>) (>A)
(>R) (>R) (>R)
(A>)
(R>) (R>) ;

: do_next_instruction
(extract_instruction)
(>R) (shift_isr) (R>)
instruction_call_table # (+) (@)
(>R) ;

: run_vm
MEM #
n 0 #
: vm_loop
c do_next_instruction
j vm_loop

```

1.5 VM4

To avoid the stack shuffling in VM3-1 (Section 1.4.5), the fourth virtual machine keeps the virtual PC and ISR in memory as PCREG and ISRREG. Combined with the trampoline mechanism, this leaves the stacks completely free of persistent extraneous data, and exactly as if the code was running natively on the hardware, with slightly shallower stacks to account for the locations temporarily used by the VM code.

In fact, functions emulating instructions which do not access memory are reduced to the instruction itself plus the trampoline indirect jump. The `mem_read` and `mem_write` code has been folded into the functions emulating instructions which access memory so as to avoid using the Return Stack. The call to `access_check` is left in place since it either leaves nothing on the stack or aborts the entire program. This method however does increase the amount of work that flow control instruction emulation functions must do, since they must interact with memory more.

```

: AREG n 1 var ;
: PCREG n 1 var ;
: ISRREG n 1 var ;

: trampoline n 1 var ;

: (jump_to_trampoline)
trampoline # c # c (@) c (>R) c ; ;

<VM Parameters (Section 1.1.2)>

: MEMHEAD MEM # ;
: MEMTAIL MEM_OUTPUT # ;

: (check_low)
MEMHEAD negate # c # c (+) ;

: (check_high)
c (negate) MEMTAIL # c # c (+) ;

: mem_in_range?
(DUP) (check_low)
(OVER) (check_high) (OR) ;

: mem_access_msg
create >$ ILLEGAL_MEMORY_ACCESS: $>c
does j cs>

: report_mem_error
c mem_access_msg
c \s c #> j \n

: access_check
c mem_in_range?
if-
  c report_mem_error
  j errcontext
else ;

: do_pcfetch
PCREG # (@)
c access_check (DUP)
(N+) 1 (A!)
(@) ISRREG # (!)
(jump_to_trampoline)

: do_call
PCREG # (@) (DUP) (N+) 1 (>R)
(@) c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
(jump_to_trampoline)

: do_ret
(R>) c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
(jump_to_trampoline)

: do_jump
PCREG # (@) (@)
c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
(jump_to_trampoline)

: do_jmpzero
if
  PCREG # (@) (N+) 1 (A!)
  (jump_to_trampoline)
else
  j do_jump

: do_jmpplus
if-
  PCREG # (@) (N+) 1 (A!)
  (jump_to_trampoline)
else
  j do_jump

: do_fetcha

```

```

AREG # (@) : do_storerplus
(DUP) MEM_INPUT # (XOR) (R>)
if (DUP) MEM_OUTPUT # (XOR)
  c access_check (@) if
  (jump_to_trampoline) c access_check
else (>R) (R!+)
  (DROP) c ># (jump_to_trampoline)
  (jump_to_trampoline) else
  (N+) 1 (>R) c #>
  (jump_to_trampoline)

: do_storea
AREG # (@)
(DUP) MEM_OUTPUT # (XOR)
if
  c access_check (!)
  (jump_to_trampoline)
else
  (DROP) c #>
  (jump_to_trampoline)

: do_fetchaplus
AREG # (@)
(DUP) MEM_INPUT # (XOR)
if
  c access_check
  (>A) (A@+) (A>)
  AREG # (!)
  (jump_to_trampoline)
else
  (N+) 1 AREG # (!) c >#
  (jump_to_trampoline)

: do_storeaplus
AREG # (@)
(DUP) MEM_OUTPUT # (XOR)
if
  c access_check
  (>A) (A!+) (A>)
  AREG # (!)
  (jump_to_trampoline)
else
  (N+) 1 AREG # (!) c #>
  (jump_to_trampoline)

: do_fetchrplus
(R>)
(DUP) MEM_INPUT # (XOR)
if
  c access_check
  (>R) (R@+)
  (jump_to_trampoline)
else
  (N+) 1 (>R) c >#
  (jump_to_trampoline)

: do_lit
PCREG # (@)
(DUP) (N+) 1 (A!)
(@)
(jump_to_trampoline)

: do_undef
n 31 COMPILE_OPCODE
(jump_to_trampoline)

: do_xor
(XOR)
(jump_to_trampoline)

: do_and
(AND)
(jump_to_trampoline)

: do_not
(NOT)
(jump_to_trampoline)

: do_twostar
(2*)
(jump_to_trampoline)

: do_twoslash
(2/)
(jump_to_trampoline)

: do_plus
(+)
(jump_to_trampoline)

: do_plusstar
(++*)
(jump_to_trampoline)

: do_dup
(DUP)
(jump_to_trampoline)

: do_drop
(DROP)
(jump_to_trampoline)

```

```

: do_over
(OVER)
(jump_to_trampoline)

: do_tor
(>R)
(jump_to_trampoline)

: do_rfrom
(R>)
(jump_to_trampoline)

: do_toa
AREG # (!)
(jump_to_trampoline)

: do_afrom
AREG # (@)
(jump_to_trampoline)

: do_nop
(NOP)
(jump_to_trampoline)

```

<Instruction Call Table (Section 1.1.3)>

```

: (shift_isr)
c (2/) c (2/) c (2/) c (2/) c (2/) ;

: (extract_instruction)
OPCODEMASK # c # c (AND) ;

: do_next_instruction
ISRREG # (@)
(DUP) (shift_isr) (A!)
(extract_instruction)
instruction_call_table # (+) (@)
j EXECUTE

: run_vm
MEM # PCREG # (!)
n 0 # ISRREG # (!)
n 123456 # AREG # (!)
l# do_next_instruction
trampoline # (!)
j do_next_instruction

```

1.5.1 VM4-1 (Trampoline Removed)

As for VM3-1 in Section 1.4.5, the trampoline mechanism has been removed, the few remaining tail-calls in the memory-accessing instruction emulations have been replaced by jumps, and the call to EXECUTE has been replaced by in-line code. Only the significantly affected functions are listed below.

As expected, this required some extra stack shuffling to move the `do_next_instruction` return address out of the way, but since PC and ISR are now in memory, the overhead is much lower.

```
: do_call
(R>)
PCREG # (@)
(DUP) (N+) 1 (>R)
(@) c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
(>R) ;

: do_ret
(R>)
(R>) c access_check (DUP)
(N+) 1 PCREG # (!)
(@) ISRREG # (!)
(>R) ;

: do_fetcha
AREG # (@)
(DUP) MEM_INPUT # (XOR)
if
c access_check (@) ;
else
(DROP) j >#

: do_storea
AREG # (@)
(DUP) MEM_OUTPUT # (XOR)
if
c access_check (!) ;
else
(DROP) j #>

: do_fetchaplus
AREG # (@)
(DUP) MEM_INPUT # (XOR)
if
c access_check
(>A) (A@+) (A>)
AREG # (!) ;
else
(N+) 1 AREG # (!) j >#

: do_storeaplus
AREG # (@)
(DUP) MEM_OUTPUT # (XOR)
if
c access_check
(>A) (A!+) (A>)
AREG # (!) ;
else
(N+) 1 AREG # (!) j #>

: do_fetchrplus
(R>) (>A) (R>)
(DUP) MEM_INPUT # (XOR)
if
c access_check
(>R) (R@+)
(A>) (>R) ;
else
(N+) 1 (>R)
(A>) (>R)
j >#

: do_storerplus
(R>) (>A) (R>)
(DUP) MEM_OUTPUT # (XOR)
if
c access_check
(>R) (R!+)
(A>) (>R) ;
else
(N+) 1 (>R)
(A>) (>R)
j #>

: do_tor
(R>) (>A)
(>R)
(A>) (>R) ;

: do_rfrom
(R>) (>A)
(R>)
(A>) (>R) ;

: do_next_instruction
ISRREG # (@)
(DUP) (shift_isr) (A!)
(extract_instruction)
instruction_call_table # (+) (@)
(>R) ;

: run_vm
MEM # PCREG # (!)
n 0 # ISRREG # (!)
n 123456 # AREG # (!)
: vm_loop
c do_next_instruction
j vm_loop
```

1.6 Execution Overhead Comparison

Table 1 compares the number of instructions executed for code run on the native Gullwing machine versus the various VMs. For the native machine, the setup code was only the Flight Extensions. For the VMs, it also included the VM code, the metacompiler, and the Flight in Flight code (Section 3.3).

The test code includes all of the above, plus a recompilation of the Extensions with some code added to exercise its functions. The difference column shows the actual number of instructions executed by the test code once those from the setup code have been subtracted.

From that, the ratios of the differences relative to the native machine are calculated, which show how many instructions were executed on average for each actual instruction in the test code. From these ratios, we can observe the effects of the implementation differences between the VMs.

Number Of Executed Instructions				
Machine	Setup	Test	Difference	Ratio
Native	4,462,579	10,441,974	5,979,395	1.00000
VM1	20,193,157	253,535,067	233,359,910	39.0273
VM2	21,081,801	288,838,769	267,756,968	44.7799
VM2-1	20,918,123	276,506,826	255,588,703	42.7449
VM3	19,467,189	218,732,738	199,265,549	33.3254
VM3-1	19,617,587	203,307,626	183,690,039	30.7205
VM4	19,120,323	219,314,594	200,194,271	33.4807
VM4-1	19,332,042	204,325,338	184,993,296	30.9385

Table 1: Number Of Executed Instructions

1.6.1 Memory Protection

The only difference between VM1 and VM2 is that the latter adds the `access_check` function to memory and flow control instruction emulation functions to prevent code run inside the VM from accessing memory or executing code outside its allocated area.

The average overhead is 5.7526 instructions. The actual number of instructions in a call to `access_check`, including all call/return overhead, adds up to 13. This means that memory and flow control instructions account for about 44% of the executed test code, which agrees with the dynamic instruction frequencies listed in [Koo89, 6.3.1]⁴.

1.6.2 Tail-Call Elimination and Inlining

VM2-1 eliminates tail-calls in a number of functions and inlines the code for `EXECUTE` in `do_next_function`, which eliminates an average of 2.035 instructions. The tail-call elimination end up counting for virtually nothing since all the runtime tail-calls are to the virtual stack push and pop functions, whose typical 15 instructions dwarf the one instruction saved by the eliminated tail-call in only ten functions.

The `do_next_instruction` function is executed for every single simulated instruction, so its impact is far greater. The original code for calling to the address of the function simulating an instruction calls `EXECUTE`, which executes two instructions to enter the called function. This function then returns to `do_next_instruction`, which immediately returns to the `run_vm` loop, for a total of five instructions.

The new code executes two instructions which effectively jump to the function. This function eventually returns but since it was not called, the return instruction returns to `run_vm` directly, the caller of `do_next_instruction`, for a total of three instructions.

The elimination of what is effectively a tail-call to every simulated instruction accounts for those two saved instructions, with the other tail-call eliminations accounting for the remaining 0.035.

⁴The dynamic (meaning, executed) instruction frequencies of stack-based software roughly divide into 25% flow control, 25% memory, 50% other. This division varies a little with each program, but is accurate enough for estimates.

1.6.3 Virtual Stacks vs. Physical Stacks

The VM3 machine eliminates the virtual stacks and instead uses the actual stacks for the program running under it, and substitutes the function calls with a trampoline mechanism (8 instructions) to free the Return Stack. Relative to VM2, this saves 11.4545 instructions on average. VM3-1 discards the trampoline and adds in the same calling mechanism as done in VM2-1 (3 instructions). This now saves 14.0594 instructions relative to VM2. This difference of 2.6049 is less than the expected 5 due to the extra stack shuffling required to move the return address to `run_vm`.

Unfortunately, removing the virtual stacks affects the amount of stack shuffling the instruction simulation functions have to perform, as can be seen by comparing the code for VM1 and VM3. Further experiments would have to be done to separate the effects. Nonetheless, the advantage of not using virtual stacks is clear.

1.6.4 Location of PC and ISR

The VM4 machine alters VM3 by removing all virtual machine state from the stack and storing it in `PCREG` and `ISRREG`. In addition, to keep the Return Stack free, it also uses the trampoline mechanism. These two changes eliminate all stack shuffling for most functions, but increase the work done by the flow control instructions. The net effect is an increase of 0.1553 average instructions relative to VM3.

VM4-1 replaces the trampoline with the eliminated tail-calls of VM2-1, which saves the expected 2.5422 instructions, but is still slightly worse than VM3-1. The point of diminishing returns seems to have been reached.

1.6.5 Instruction Extraction

Throughout all the virtual machines, the single largest source of overhead is the process of extracting the next instruction. The `do_next_instruction` function takes from 16 to 19 instructions to execute, depending on the VM. This is more than the instructions saved by all the previously mentioned optimizations. This overhead is irreducible and is the main reason for the poor performance of virtual machines on the Gullwing processor.

2 Virtualization Requirements

The Popek and Goldberg virtualization requirements [PG74] can be used to explain the sources of overhead in the virtual machines (VMs) and suggest changes to the Gullwing architecture to reduce them.

2.1 Virtual Machine Properties

The requirements assume three major properties of a virtual machine⁵:

1. *Equivalence*: a program runs identically on the virtual machine than on the native one.
2. *Resource Control*: the virtual machine must be in complete control of the allocated resources.
3. *Efficiency*: the majority of the executed instructions do so directly on the native machine.

The virtual machines presented in Section 1) meet the first criterion, with the exception of the lack of address translation. If a program uses hardcoded addresses such as the location of the native I/O ports, it will behave differently in a virtual machine.

The second criterion is met without any reservations, since all flow control and memory access instructions are bounds-checked.

The VMs completely fail the third criterion, since none of the program instructions execute natively. This is the cause of all the overhead mentioned in Section 1.6 and the main weakness addressed by this section.

2.2 Instruction Set Architecture Properties

With the assumption that the native machine has two modes, *user* and *supervisor*, and a means to trap between them, the instruction set is broken down into three groups:

1. *Privileged*: instructions that trap when executed in user mode, but not in supervisor mode.
2. *Control Sensitive*: instructions that can change the control flow or the allocated resources.
3. *Behaviour Sensitive*: instructions whose behaviour depends on the configuration of the resources.

The Gullwing processor has no modes, and so has no privileged instructions. The instructions which alter the program control flow (CALL, JMP, RET, etc...) and the instructions which access memory (A@, R!+, LIT, etc...) are all control sensitive. There are no behaviour sensitive instructions.

2.3 Theorem

The main theorem proven in [PG74] is as follows:

For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

Summarized, a third generation computer is one with a code relocation mechanism (address translation and bounds-checking), user/supervisor modes, and a trap mechanism. The Gullwing processor has none of these and so immediately fails to meet the specifications of the theorem, but this is put aside for now and addressed in the next subsection. More importantly, the Gullwing processor is not virtualizable since its set of privileged instructions is empty, and thus the non-empty set of sensitive instructions cannot be its subset.

2.4 Making Gullwing Virtualizable

The way to make the Gullwing processor virtualizable then is to alter it such that the sensitive instructions become privileged instructions. This requires at least user and supervisor modes and a trap mechanism which will trigger whenever an attempt is made to execute a sensitive instruction while in user mode.

One form of such a mechanism would place the contents of the Instruction Shift Register (ISR) on the Data Stack, place the contents of the Program Counter (PC) on the Return Stack, switch to supervisor mode, and jump to the trap handler. The handler

⁵Strictly speaking, the properties are those of a Virtual Machine Monitor, which can control multiple virtual machines, but only the singular case is considered here.

would then extract the instruction from the ISR, look it up in a table, call its emulating function, and then simultaneously switch to user mode, reload the ISR and PC, and return to the program using a “return from trap” instruction⁶.

A relocation register is not strictly required. If the trapping mechanism includes the instructions which access memory, then the trap handler can do the address translation and bounds-checking in software. This is a weaker form of virtualization since it does fulfil the efficiency property as much (Section 2.1), but the majority of the executed instructions will still do so directly. This is the approach taken for now, for simplicity.

2.5 A Hypothetical VM5

Most of the instructions simulated in VM4 (Section 1.5) execute natively, wrapped only in a trampoline (Section 1.4.3), which is not far from the ideal VM mentioned in Section 2.1. The source for VM4 can be used as the basis for a sketch of a hypothetical, improved virtual machine on the virtualizable version of the Gullwing processor outlined in the previous subsection.

This virtual machine still suffers from the need to extract instructions from the ISR (Section 1.6.5), but now does so only for the privileged instructions, which count for less than 50% (Section 1.6.1) of the executed total.

2.5.1 Final Improvement

Adding a relocation register and limiting the traps on the privileged instructions to memory range traps would effectively implement in hardware the function of `access_check` and its subroutines. This would invoke the trap handler, and its associated instruction extraction overhead, only if an attempt is made to pass program control or access a memory location outside of the bounds specified by the relocation register. This would eliminate all emulation code, reduce the trap handler to dealing with exceptional cases, and leave a well-behaved program running unimpeded on the native machine.

2.5.2 Source Code Draft

```
<VM Parameters (Section 1.1.2)>

: MEMHEAD MEM # ;
: MEMTAIL MEM_OUTPUT # ;

: (check_low)
MEMHEAD negate # c # c (+) ;

: (check_high)
c (negate) MEMTAIL # c # c (+) ;

: mem_in_range?
(DUP) (check_low)
(OVER) (check_high) (OR) ;

: mem_access_msg
create >$ ILLEGAL_MEMORY_ACCESS: $>c
does j cs>

: report_mem_error
c mem_access_msg
c \s c #> j \n

: access_check
c mem_in_range?
if-
  c report_mem_error
  j errcontext
else ;

: do_pcfetch
(R>) (R>)
c access_check
(>R) (>R) ;

: do_call
(R>) (R>) (DUP)
(>R) (>R) (@)
c access_check
(DROP) ;

: do_ret
(R>) (R>)
c access_check
(>R) (>R) ;

: do_jump
(R>) (R>) (DUP)
(>R) (>R) (@)
c access_check
(DROP) ;

: do_jumpzero
(R>) (R>) (DUP)
(>R) (>R) (@)
c access_check
(DROP) ;

: do_jmpplus
(R>) (R>) (DUP)
```

⁶If it switched to user mode before the return, the return would trap, leading to an infinite loop. Thus a privileged “return from trap” instruction is required.

```

(>R) (>R) (@)
c access_check
(DROP) ;

: do_fetcha
(get_A_register)
c access_check
(DROP) ;

: do_storea
(get_A_register)
c access_check
(DROP) ;

: do_fetchaplus
(get_A_register)
c access_check
(DROP) ;

: do_storeaplus
(get_A_register)
c access_check
(DROP) ;

```

```

: do_fetchrplus
(R>) (R>) (R>)
c access_check
(>R) (>R) (>R) ;

: do_storerplus
(R>) (R>) (R>)
c access_check
(>R) (>R) (>R) ;

```

```

: do_lit
(R>) (R>)
c access_check
(>R) (>R) ;

: , c #>c c ALIGN ;
: &> c l c , ;

: instruction_call_table
create
&> do_pcfetch
&> do_call
&> do_ret
&> do_jump
&> do_jumpzero
&> do_jumpplus
&> do_fetcha
&> do_storea
&> do_fetchaplus
&> do_storeaplus
&> do_fetchrplus
&> do_storerplus
&> do_lit
does ;

: (shift_isr)
c (2/) c (2/) c (2/) c (2/) c (2/) ;

: (extract_instruction)
OPCODEMASK # c # c (AND) ;

: trap_handler
(save_A_register)
(extract_instruction)
(>R) (shift_isr) (R>)
instruction_call_table # (+) (@)
c EXECUTE
(restore_A_register)
(return_from_trap)

```

3 Metacompiler

A limitation of Flight is that it can only compile as an extension of itself. Names and code must end up in their respective dictionaries. This makes it impossible to write code for other locations in memory, such as the MEM of a virtual machine, short of implementing an assembler for the virtual machine and starting over from scratch.

However, by saving, altering, and restoring the dictionary pointers a kind of new 'context' can be created which will transparently point Flight to different name and code dictionaries, and with a little cleverness, keep the ability to lookup and use the code in the original dictionaries!

Borrowing a term from Forth linguo, this technique is called *metacompilation*, since it allows the full power of the language to be used to create a separate piece of software, possibly including a new copy of itself⁷.

Some trivial modifications to the Kernel and the Extensions are required. They are documented in Appendices A and B. The purpose of the pointers is described in [LaF05c] and [LaF05b], and summarized as follows:

HERE	Current location at which code is being compiled
HERE_NEXT	Next location at which code will be compiled
THERE	Location at which the next name will be compiled
SLOT	Current instruction slot available in a memory word
INPUT	Location of the head of the input buffer
NAME_END	Location of the tail of the name dictionary

3.1 Limitations

The biggest limitation of this metacompiler is that it cannot translate addresses. It is not possible to compile code that is meant to run at a different address than the one at which it was compiled, which is the actual memory location of the target (Section 1). For example, this makes it impossible to compile code that would be burnt to ROM (Read-Only Memory) or for virtual machines which do address translation.

Translating addresses in a VM at run-time consists of simply adding an offset to the memory addresses used by fetches and stores in the program being run inside. Since the metacompiler uses Flight's own compilation code to generate the target, the capability to add an offset would have to be built into the Flight Kernel itself, and it is not obvious where to do so short of a complete rewrite⁸, nor is it a required capability for its basic operation.

In the worst case, the non-translating metacompiler could be used to compile a translating version of Flight run inside a VM, in the same manner as the one in Section 3.3. The metacompiler would then be compiled under this new Flight, and then used to create an address-translated target.

3.2 Source

These are used to save the target location of Flight.

```
: vm_here n 1 var ;
: vm_here_next n 1 var ;
: vm_there n 1 var ;
: vm_slot n 1 var ;
: vm_input n 1 var ;
: vm_name_end n 1 var ;
```

These are used to save the original state of Flight.

```
: nm_here n 1 var ;
: nm_here_next n 1 var ;
: nm_there n 1 var ;
: nm_slot n 1 var ;
: nm_input n 1 var ;
: nm_name_end n 1 var ;
```

The following functions simply save and restore the original and target values.

```
: save_nm_here l# HERE (@) nm_here # (!) ;
: save_nm_here_next l# HERE_NEXT (@) nm_here_next # (!) ;
: save_nm_slot l# SLOT (@) nm_slot # (!) ;
: save_nm_input l# INPUT (@) nm_input # (!) ;
: save_nm_there l# THERE (@) nm_there # (!) ;
: save_nm_name_end l# NAME_END (@) nm_name_end # (!) ;
```

⁷Strictly speaking, this is barely a metacompiler. Real metcompilers can cross-compile to other platforms, test the foreign code, and alter compiled addresses when compiling code at a different location than the one at which it is intended to run. They are extremely arcane pieces of software, with very little widely published about them.

⁸It would be strange if Flight could not not be extended to support address translation. Hopefully only the lower-level compiling words, such as `CMPCALL`, would need to be redefined.

```

: save_vm_here l# HERE (@) vm_here # (!) ;
: save_vm_here_next l# HERE_NEXT (@) vm_here_next # (!) ;
: save_vm_slot l# SLOT (@) vm_slot # (!) ;
: save_vm_input l# INPUT (@) vm_input # (!) ;
: save_vm_there l# THERE (@) vm_there # (!) ;
: save_vm_name_end l# NAME_END (@) vm_name_end # (!) ;

: restore_nm_here nm_here # (@) l# HERE (!) ;
: restore_nm_here_next nm_here_next # (@) l# HERE_NEXT (!) ;
: restore_nm_slot nm_slot # (@) l# SLOT (!) ;
: restore_nm_input nm_input # (@) l# INPUT (!) ;
: restore_nm_there nm_there # (@) l# THERE (!) ;
: restore_nm_name_end nm_name_end # (@) l# NAME_END (!) ;

: restore_vm_here vm_here # (@) l# HERE (!) ;
: restore_vm_here_next vm_here_next # (@) l# HERE_NEXT (!) ;
: restore_vm_slot vm_slot # (@) l# SLOT (!) ;
: restore_vm_input vm_input # (@) l# INPUT (!) ;
: restore_vm_there vm_there # (@) l# THERE (!) ;
: restore_vm_name_end vm_name_end # (@) l# NAME_END (!) ;

: init_vm_here MEM # vm_here # (!) n# 0 (A@) (!) ;
: init_vm_here_next MEM n 1 + # vm_here_next # (!) n# 0 (A@) (!) ;
: init_vm_slot n# 0 vm_slot # (!) ;
: init_vm_there MEM_INPUT n 1 - # vm_there # (!) ;
: init_vm_input MEM_INPUT # vm_input # (!) ;
: init_vm_name_end MEM_INPUT # vm_name_end # (!) ;

```

The `vm_nxec` function is the core of the metacompiler. Like the original `NXEC` in `Flight`, it reads in a string, looks up the address of the associated code, and calls to it. It does so first in the target name dictionary, and if the name is not found there, it swaps the name dictionary pointers with those of the original one and looks there. It then swaps the pointers again and executes the function.

This allows the use of `Flight` to bootstrap an empty target by using its existing compiling functions. Furthermore, if a target function is compiled with the same name as an original `Flight` function, then it will get found instead. Later references to the function will now remain inside the target code. With care, the target can be completely 'weaned' from references to the original `Flight` code.

```

c restore_nm_name_end
c LOOK
c save_nm_there
c save_nm_name_end
c restore_vm_there
c restore_vm_name_end
c POP_STRING
c EXECUTE
j vm_nxec

```

The `LOOK` function expects a non-empty name dictionary. This function initializes a name dictionary by creating a dummy entry called 'zeroword' that points to address 0.

```

: vm_nxec
c SCAN
c LOOK
(DUP) l# NAME_END (@) (XOR)
if
c POP_STRING
c EXECUTE
j vm_nxec
else
(DROP)
c save_vm_there
c save_vm_name_end
c restore_nm_there

: init_name_dict
create >$ zeroword $>c
does c c>$ n# 0 c DEFN_AS ;

: (unwind) c (R>) c (DROP) ;

```

The `(unwind)` function discards the address of the calling function. It is used when the control flow of a program is forcibly changed and prevents the old addresses from filling up the Return Stack.

The VM (function initializes the target state and swaps the original state for it. It then initializes the target name dictionary and substitutes vm_exec as the main loop.

```

: VM(
c init_vm_here
c init_vm_here_next
c init_vm_slot
c init_vm_there
c init_vm_input
c init_vm_name_end
c save_nm_here
c save_nm_here_next
c save_nm_slot
c save_nm_there
c save_nm_input
c save_nm_name_end
c restore_vm_here
c restore_vm_here_next
c restore_vm_slot
c restore_vm_there
c restore_vm_input
c restore_vm_name_end
c init_name_dict
(unwind)
j vm_nxec

```

The) VM function does the opposite, returning the state and main loop to normal.

```

: )VM
c save_vm_here
c save_vm_here_next
c save_vm_slot
c save_vm_there
c save_vm_input
c save_vm_name_end
c restore_nm_here
c restore_nm_here_next
c restore_nm_slot
c restore_nm_there
c restore_nm_input
c restore_nm_name_end
(unwind)
j NXEC

```

3.3 Flight In Flight

As an example of metacompilation, here is the Flight Kernel implemented in Flight code and targeted into the MEM of a virtual machine. The object code generated is virtually identical to the original Kernel⁹.

```

VM(
n 0 (JMP) MEM n 1 + >$ START_WORD DEFN_AS

: HERE
: HERE_NEXT
: THERE
: SLOT
: INPUT
: NAME_END

: MINUS (NOT) n# 1 (+) (+) ;
: OR (OVER) (NOT) (AND) (XOR) ;

: READ1 MEM_INPUT # (@) ;
: WRITE1 MEM_OUTPUT # (!) ;

: STRING_TAIL (>A) (A@+) (A>) (+) ;

: PUSH_STRING
(DUP) n# 1 (+) (NOT)
l# INPUT (@) (+) (DUP) (A!)
(!)
l# INPUT (@)

c STRING_TAIL
(DUP) (!) ;

: POP_STRING
l# INPUT (@)
c STRING_TAIL
n# 1 (+)
l# INPUT (!) ;

: COMPARE_STRINGS
(>A) (>R)
: CS_LOOP
(R@+) (A@+) (XOR)
j0 CS_LOOP
-n# 1 (R>) (+)
-n# 1 (A>) (+) ;

: SCAN_STRING
l# INPUT (@) (>R) (R@+)
: SS_LOOP
c READ1 (R!+)
-n# 1 (+) (DUP)
if j SS_LOOP
else (DROP) (R>) (DUP) (!) ;

```

⁹A few memory locations are wasted, and this code contains the bug-fixes and improvements done since the original Kernel.

```

: SCAN
c READ1
c PUSH_STRING
j SCAN_STRING

: FIRST_SLOT
l# SLOT (>A)
n# 31 (A!) ;

: LAST_SLOT
l# SLOT (>A)
n# 1040187392 (A!) ;

: NULL_SLOT
l# SLOT (>A)
n# 0 (A!) ;

: ALIGN
l# HERE_NEXT (@)
(DUP) (>R) n# 0 (R!+)
(R>) (A!)
l# HERE (!)
j NULL_SLOT

: NEXT_SLOT
l# SLOT (@)
n# 0 (OVER) (XOR)
if
  n# 1040187392 (OVER) (XOR)
  if
    (2*) (2*) (2*) (2*) (2*) (A!) ;
  else
    (DROP) c ALIGN j FIRST_SLOT
else
  (DROP) j FIRST_SLOT

: DEFN_AS
l# THERE (@) (!)
l# INPUT (@)
-n# 1 (+)
l# THERE (!) ;

: NEW_WORD
c ALIGN
l# HERE (@) ;

: DEFN
c NEW_WORD
j DEFN_AS

: LOOK
l# THERE (@) n# 1 (+) (DUP)
: LOOK_LOOP
l# INPUT (@)
c COMPARE_STRINGS
l# INPUT (@)

c STRING_TAIL (XOR)
if
  (DROP) c STRING_TAIL
  n# 1 (+)
  (DUP) l# NAME_END (@) (XOR)
  if
    (DUP) j LOOK_LOOP
  else
    (DROP) l# NAME_END (@) ;
else
  (>A) (DROP) (A@) ;

: COMPILE_OPCODE
c NEXT_SLOT
(>R) l# SLOT (@) (NOT)
(R>) (OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
(OVER) (OVER) (AND) if
(2*) (2*) (2*) (2*) (2*)
else else else else else
l# HERE (@)
(@) (+) (A!) (DROP) ;

: COMPILE_LITERAL
l# HERE_NEXT (@)
(>A) (A!+)
(A>) l# HERE_NEXT (!) ;

: CMPCALL
CALLopcode #
c COMPILE_OPCODE
c COMPILE_LITERAL
j ALIGN

: CMPJMP
JMPopcode #
c COMPILE_OPCODE
c COMPILE_LITERAL
j ALIGN

: CMPJMPZERO
JMPZEROopcode #
c COMPILE_OPCODE
j COMPILE_LITERAL

: CMPJMPPLUS
JMPPLUSopcode #
c COMPILE_OPCODE
j COMPILE_LITERAL

```

```

: NUMC
LITopcode #
c COMPILE_OPCODE
j COMPILE_LITERAL

: CMPPCFETCH
PCFETCHopcode #
j COMPILE_OPCODE

: CMPRET
RETopcode #
j COMPILE_OPCODE

: CMPFETCHPLUS
FETCHPLUSopcode #
j COMPILE_OPCODE

: CMPFETCHRPLUS
FETCHRPLUSopcode #
j COMPILE_OPCODE

: CMPFETCHA
FETCHAopcode #
j COMPILE_OPCODE

: CMPSTOREAPLUS
STOREAPLUSopcode #
j COMPILE_OPCODE

: CMPSTORERPLUS
STORERPLUSopcode #
j COMPILE_OPCODE

: CMPSTOREA
STOREAopcode #
j COMPILE_OPCODE

: CMPNOT
NOTopcode #
j COMPILE_OPCODE

: CMPAND
ANDopcode #
j COMPILE_OPCODE

: CMPXOR
XORopcode #
j COMPILE_OPCODE

: CMPPLUS
PLUSopcode #
j COMPILE_OPCODE

: CMPTWOSTAR
TWOSTARopcode #
j COMPILE_OPCODE

: CMPTWOSLASH
TWOSLASHopcode #
j COMPILE_OPCODE

: CMPPLUSSTAR
PLUSSTARopcode #
j COMPILE_OPCODE

: CMPAFROM
AFROMopcode #
j COMPILE_OPCODE

: CMPTOA
TOAopcode #
j COMPILE_OPCODE

: CMPDUP
DUPopcode #
j COMPILE_OPCODE

: CMPDROP
DROPOpcode #
j COMPILE_OPCODE

: CMPOVER
OVERopcode #
j COMPILE_OPCODE

: CMPTOR
TORopcode #
j COMPILE_OPCODE

: CMPRFROM
RFROMopcode #
j COMPILE_OPCODE

: CMPNOP
NOPopcode #
j COMPILE_OPCODE

: EXECUTE
(>R) ;

: EXEC
c READ1
c EXECUTE
j EXEC

: NEXEC
c SCAN
c LOOK
c POP_STRING
c EXECUTE
j NEXEC

: TENSTAR
(DUP) (2*) (2*) (2*) (OVER) (+) (+) ;

```

```

: NUMI
l# INPUT (@)
(>R) (R@+)
n# 0 (>A)
(DUP)
if
: NUMI_LOOP
(A>) c TENSTAR (R@+) -n# 48 (+) (+) (>A)
-n# 1 (+) (DUP)
if
j NUMI_LOOP
else
else
(DROP) (R>) (DROP) (A>)
j POP_STRING
l NEXEC l START_WORD !
Before the target is ready for execution, the values of its pointers saved in the metacompiler must be placed into their target equivalents.
save_vm_here vm_here @ l HERE !
save_vm_here_next vm_here_next @ l HERE_NEXT !
save_vm_slot vm_slot @ l SLOT !
save_vm_input vm_input @ l INPUT !
save_vm_there vm_there @ l THERE !
save_vm_name_end vm_name_end @ l NAME_END !
)VM

```

A Flight Kernel Changes (v. 6)

Core Words Of Flight - Take 6

2006/03/02

Altered LOOK to not pop the string on the input stack and to return the value of NAME-END if no match so we can test against that for error checking. This means changing some low-level words in the Flight extensions so as to keep things working as before since most of the time the input string doesn't need to be kept.

2006/02/25

Altered LOOK to use NAME-END as end of dictionary address, so it can be temporarily altered for retargeted compilation.

2006/01/05

Fixed PUSH-STRING to set the tail

B Flight Extension Changes

```

SCAN : DEFN
SCAN SCAN LOOK POP_STRING CMPCALL
SCAN DEFN LOOK POP_STRING CMPCALL CMPRET

: l
SCAN SCAN LOOK POP_STRING CMPCALL
SCAN LOOK LOOK POP_STRING CMPCALL
SCAN POP_STRING LOOK POP_STRING CMPCALL CMPRET

: $c
l LOOK CMPCALL
l POP_STRING CMPCALL
l CMPCALL CMPCALL CMPRET

alias POP_STRING $pop

: $j c $l c $pop c (JMP) ;
: $j0 c $l c $pop c (JMP0) ;
: $j+ c $l c $pop c (JMP+) ;

```

References

- [Koo89] Philip J. Koopman, *Stack computers: the new wave*, Halsted Press, 1989.
- [LaF05a] Eric LaForest, *Unit 2-3 (IS 102B) report: The Gullwing stack computer architecture*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.
- [LaF05b] _____, *Unit 2-3 (IS 301B) report: Extensions to the Flight programming language*, IS Unit report, University of Waterloo, Independent Studies Program, November 2005, Otherwise unpublished.
- [LaF05c] _____, *Unit 4-5 (IS 101B) report: The Flight programming language*, IS Unit report, University of Waterloo, Independent Studies Program, April 2005, Otherwise unpublished.
- [LaF05d] _____, *Unit 4-5 (IS 302B) Burst-Mode locally-clocked asynchronous implementation of the gullwing stack computer architecture*, IS Unit report, University of Waterloo, Independent Studies Program, December 2005, Otherwise unpublished.
- [PG74] Gerald J. Popek and Robert P. Goldberg, *Formal requirements for virtualizable third generation architectures*, Commun. ACM **17** (1974), no. 7, 412–421.