Gambit Compiler Pipeline

Scheme

(define (main)
  (println (f 9)))

(define (f x)
  (square (fx+ x 1)))

(define (square y)
  (fx* y y))

GVM

**** #<procedure main> =
  #1 fs=0 entry-point nparams=0 ()
  frame[1] = r0
  r1 = '9
  jump fs=4 global[f] r0=#2 nargs=1
  #2 fs=4 return-point
  r0 = frame[1]
  jump fs=0 global[println] nargs=1

**** #<procedure f> =
  #1 fs=0 entry-point nparams=1 ()
  r1 = (fx+ r1 '1)
  jump fs=0 global[square] nargs=1

**** #<procedure square> =
  #1 fs=0 entry-point nparams=1 ()
  r1 = (fx* r1 r1)
  jump fs=0 r0

C backend

C code

JavaScript code

JS backend

frontend
Gambit Compiler Pipeline

Scheme

(define (main)
 (println (f 9)))

(define (f x)
 (square (fx+ x 1)))

(define (square y)
 (fx* y y))

GVM

**** #<procedure main> =
#1 fs=0 entry-point nparams=0 ()
 frame[1] = r0
 r1 = '9
 jump fs=4 global[f] r0=#2 nargs=1
#2 fs=4 return-point
 r0 = frame[1]
 jump fs=0 global[println] nargs=1

**** #<procedure f> =
#1 fs=0 entry-point nparams=1 ()
 r1 = (fx+ r1 '1)
 jump fs=0 global[square] nargs=1

**** #<procedure square> =
#1 fs=0 entry-point nparams=1 ()
 r1 = (fx* r1 r1)
 jump fs=0 r0

C backend

C code

JavaScript code

JS backend

Native backend (x86-32/64, ARM)

Native code

NEW!
C/Native Interoperability

Separate compilation of modules by C and Native backends

Scheme

```
(define (main)
  (println (f 9)))

(define (f x)
  (square (fx+ x 1)))
```

GVM

```
**** #<procedure main> =
  #1 fs=0 entry-point nparams=0 ()
  frame[1] = r0
  r1 = '9
  jump fs=4 global[f] r0=#2 nargs=1
  #2 fs=4 return-point
  r0 = frame[1]
  jump fs=0 global[println] nargs=1

**** #<procedure f> =
  #1 fs=0 entry-point nparams=1 ()
  r1 = (fx+ r1 '1)
  jump fs=0 global[square] nargs=1

**** #<procedure square> =
  #1 fs=0 entry-point nparams=1 ()
  r1 = (fx* r1 r1)
  jump fs=0 r0
```

C backend

C code

Native backend

(x86-32/64, ARM)

Native code
C/Native Interoperability

Scheme

(define (main)
  (println (f 9)))

(define (f x)
  (square (fx+ x 1)))

(define (square y)
  (fx* y y))

GVM

**** #<procedure main> =
#1 fs=0 entry-point nparams=0 ()
  frame[1] = r0
  r1 = '9
  jump fs=4 global[f] r0=#2 nargs=1
#2 fs=4 return-point
  r0 = frame[1]
  jump fs=0 global[println] nargs=1

**** #<procedure f> =
#1 fs=0 entry-point nparams=1 ()
  r1 = (fx+ r1 '1)
  jump fs=0 global[square] nargs=1

**** #<procedure square> =
#1 fs=0 entry-point nparams=1 ()
  r1 = (fx* r1 r1)
  jump fs=0 r0

C backend

C code

CROSS JUMPS

Native backend

(x86-32/64, ARM)

Native code
C/Native Interoperability

**Scheme**

```
(define (main)
  (println (f 9)))
(define (f x)
  (square (fx+ x 1)))
(define (square y)
  (fx* y y))
```

**GVM**

```
**** #<procedure main> =
   #1 fs=0 entry-point nparams=0 ()
   frame[1] = r0
   r1 = '9
   jump fs=4 global[f] r0=#2 nargs=1
   #2 fs=4 return-point
   r0 = frame[1]
   jump fs=0 global[println] nargs=1

**** #<procedure f> =
   #1 fs=0 entry-point nparams=1 ()
   r1 = (fx+ r1 '1)
   jump fs=0 global[square] nargs=1

**** #<procedure square> =
   #1 fs=0 entry-point nparams=1 ()
   r1 = (fx* r1 r1)
   jump fs=0 r0
```

**C backend**

C code

**Native backend**

(x86-32/64, ARM)

**How?**

CROSS JUMPS

frontend

frontend

frontend
Why?

• Reuse runtime library of C backend
  • Complex C functions (garbage collector, OS interface, …)
  • Complex Scheme library procedures (eval, bignums, …)
  • Complex I/O procedures (pretty-print, read, …)
  • Simple primitives, like ##cons and ##flsqr that are not yet inlined by Native backend (useful during development of backend)
• Mix modules compiled by C and Native backends and compile performance critical parts with Native backend
Desiderata

1. **Cross jumps are transparent** (don’t know ahead of time if a jump is going to be a cross jump or a local jump)

2. **Native code jumps are plain branch-to-address instructions** (no overhead on native local jumps due to support for cross jumps)

3. **Cost of cross jumps is similar to C backend local jumps** (which are performed with a trampoline)
C Backend

- Uses a **processor state structure** to store the state of the Gambit Virtual Machine (registers, program counter, ...)

```c
typedef struct processor_state_struct {
    WORD r0, r1, r2, r3, r4;    /* registers */
    WORD *fp;                    /* frame ptr */
    WORD *hp;                    /* heap ptr */
    WORD *pc;                    /* program ctr */
    ...
} processor_state_struct;
```

- Uses a **label structure** to represent control points in the code (either procedure entry point or return point)

```c
typedef struct label_struct {
    WORD HEADER;
    WORD entry;
    WORD unused;
    void (*host)(processor_state_struct *ps);
} label_struct;
```
C Backend

• Uses a **processor state structure** to store the state of the Gambit Virtual Machine (registers, program counter, ...)

```c
typedef struct processor_state_struct {
    WORD r0, r1, r2, r3, r4;   /* registers */
    WORD *fp;                  /* frame ptr */
    WORD *hp;                  /* heap ptr */
    WORD *pc;                  /* program ctr */
    ...
} processor_state_struct;
```

• Uses a **label structure** to represent control points in the code (either procedure entry point or return point)

```c
typedef struct label_struct {
    WORD HEADER;
    WORD entry;
    WORD unused;
    void (*host)(processor_state_struct *ps);
} label_struct;
```

`pc` points to the currently executing control point.
C Backend

- Uses a **processor state structure** to store the state of the Gambit Virtual Machine (registers, program counter, ...)

```c
typedef struct processor_state_struct {
    WORD r0, r1, r2, r3, r4; /* registers */
    WORD *fp; /* frame ptr */
    WORD *hp; /* heap ptr */
    WORD *pc; /* program ctr */
    ...
} processor_state_struct;
```

- Uses a **label structure** to represent control points in the code (either procedure entry point or return point)

```c
typedef struct label_struct {
    WORD HEADER;
    WORD entry;
    WORD unused;
    void (*host)(processor_state_struct *ps);
} label_struct;
```

- **pc** points to the currently executing control point
- The **host** function is the C function containing this control point
Example

(\(\text{define } (\text{main}) (\text{println } (f \ 9))\))

(\(\text{define } (f \ x) (\text{square } (fx+ \ x \ 1))\))

(\(\text{define } (\text{square } y) (fx* \ y \ y)\))
Example

(define (main) (println (f 9)))

(define (f x) (square (fx+ x 1)))

(define (square y) (fx* y y))

Non-tail call

Tail call
void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2; /* tagged fixnum = 9 */
                 ps->r0 = 1+(WORD)&labels[1]; /* tagged return address */
                 ps->pc = glo_f; /* non-tail call f */
                 return;
        case 1: printf("%d\n", ps->r1 >> 2);
                 exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[2]))) / sizeof(label_struct)) {
        case 0: ps->r1 += 4; /* add 1 to arg */
                 ps->pc = glo_square; /* tail call square */
                 return;
    }
}

void host_square(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[3]))) / sizeof(label_struct)) {
        case 0: ps->r1 *= (ps->r1 >> 2); /* square arg */
                 ps->pc = ps->r0; /* return to main */
                 return;
    }
}
void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2; /* tagged fixnum = 9 */
            ps->r0 = 1+(WORD)&labels[1]; /* tagged return address */
            ps->pc = glo_f; /* non-tail call f */
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[2]))) / sizeof(label_struct)) {
        case 0: ps->r1 += 4; /* add 1 to arg */
            ps->pc = glo_square; /* tail call square */
            return;
    }
}

void host_square(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[3]))) / sizeof(label_struct)) {
        case 0: ps->r1 *= (ps->r1 >> 2); /* square arg */
            ps->pc = ps->r0; /* return to main */
            return;
    }
}
void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct)) {

        case 0:
            ps->r1 = 9 << 2;
            ps->r0 = 1+(WORD)&labels[1];
            ps->pc = glo_f;
            return;

        case 1:
            printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

ps->r1 = 9 << 2;
ps->r0 = 1+(WORD)&labels[1];
ps->pc = glo_f;

(define (main) (println (f 9)))
(define (f x) (square (fx+ x 1)))
void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0])) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2;
            ps->r0 = 1+(WORD)&labels[1];
            ps->pc = glo_f;
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

Return address for call to f refers to second basic block

ps->r1 = 9 << 2;
ps->r0 = 1+(WORD)&labels[1];
ps->pc = glo_f;

(define (main) (println (f 9)))
(define (f x) (square (fx+ x 1)))
Control Points

```c
label_struct labels[] = {
    { ..., ..., ..., host_main } /* 0 */
, { ..., ..., ..., host_main } /* 1 */
, { ..., ..., ..., host_f } /* 2 */
, { ..., ..., ..., host_square } /* 3 */
};

WORD glo_main = 1+(WORD)&labels[0];
WORD glo_f = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];
```
Procedures and return addresses are tagged

Return addresses are first-class objects in Gambit Scheme (allows continuation inspection in Scheme and simplifies GC)
void trampoline(processor_state_struct *ps) {
    while (TRUE) GET_HOST(ps->pc)(ps);
}

void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2;
            ps->r0 = 1+(WORD)&labels[1];  /* tagged return address */
            ps->pc = glo_f;
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[2]))) / sizeof(label_struct)) {
        case 0: ps->r1 += 4;
            ps->pc = glo_square;
            return;
    }
}

void host_square(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[3]))) / sizeof(label_struct)) {
        case 0: ps->r1 *= (ps->r1 >> 2);
            ps->pc = ps->r0;
            return;
    }
}
Execution starts with call to `trampoline` with `ps->pc` referring to program entry point.
void trampoline(processor_state_struct *ps) {
    while (TRUE) GET_HOST(ps->pc)(ps);
}

label_struct labels[] = {
    { ..., ..., ..., host_main  },
    { ..., ..., ..., host_main  },
    { ..., ..., ..., host_f     },
    { ..., ..., ..., host_square}
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];

void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2; /* tagged fixnum = 9 */
            ps->r0 = 1+(WORD)&labels[1]; /* tagged return address */
            ps->pc = glo_f; /* non-tail call f */
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[2]))) / sizeof(label_struct)) {
        case 0: ps->r1 += 4; /* add 1 to arg */
            ps->pc = glo_square; /* tail call square */
            return;
    }
}

void host_square(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[3]))) / sizeof(label_struct)) {
        case 0: ps->r1 *= (ps->r1 >> 2); /* square arg */
            ps->pc = ps->r0; /* return to main */
            return;
    }
}
void trampoline(processor_state_struct *ps) {
    while (TRUE) GET_HOST(ps->pc) (ps);
}

void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2; /* tagged fixnum = 9 */
            ps->r0 = 1+(WORD)&labels[1]; /* tagged return address */
            ps->pc = glo_f; /* non-tail call f */
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[2]))) / sizeof(label_struct)) {
        case 0: ps->r1 += 4; /* add 1 to arg */
            ps->pc = glo_square; /* tail call square */
            return;
    }
}

void host_square(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)&labels[3]))) / sizeof(label_struct)) {
        case 0: ps->r1 *= (ps->r1 >> 2); /* square arg */
            ps->pc = ps->r0; /* return to main */
            return;
    }
}

label_struct labels[] = {
    …,
    { …, …, …, host_main },
    { …, …, …, host_main },
    { …, …, …, host_f },
    { …, …, …, host_square }
};

WORD glo_main   = 1+(WORD)&labels[0];
WORD glo_f      = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];

ps->r0
ps->pc
```c
void trampoline(processor_state_struct *ps) {
    while (TRUE) GET_HOST(ps->pc)(ps);
}

void host_main(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)\&labels[0]))) / sizeof(label_struct)) {
        case 0: ps->r1 = 9 << 2; /* tagged fixnum = 9 */
            ps->r0 = 1+(WORD)\&labels[1]; /* tagged return address */
            ps->pc = glo_f; /* non-tail call f */
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)\&labels[2]))) / sizeof(label_struct)) {
        case 0: ps->r1 += 4; /* add 1 to arg */
            ps->pc = glo_square; /* tail call square */
            return;
    }
}

void host_square(processor_state_struct *ps) {
    switch ((ps->pc - (1+(WORD)\&labels[3]))) / sizeof(label_struct)) {
        case 0: ps->r1 *= (ps->r1 >> 2); /* square arg */
            ps->pc = ps->r0; /* return to main */
            return;
    }
}

label_struct labels[] = {
    { ..., ..., ..., host_main          },
    { ..., ..., ..., host_main         } ← ps->r0,
    { ..., ..., ..., host_f            } ← ps->pc,
    { ..., ..., ..., host_square       } ← ps->pc
};

WORD glo_main   = 1+(WORD)\&labels[0];
WORD glo_f      = 1+(WORD)\&labels[2];
WORD glo_square = 1+(WORD)\&labels[3];
```

```
void trampoline(processor_state_struct *ps) {
    while (TRUE) GET_HOST(ps->pc)(ps);
}

label_struct labels[] = {
    { ..., ..., ..., host_main }
    , { ..., ..., ..., host_main }
    , { ..., ..., ..., host_f }
    , { ..., ..., ..., host_square }
};

WORD glo_main = 1+(WORD)&labels[0];
WORD glo_f = 1+(WORD)&labels[2];
WORD glo_square = 1+(WORD)&labels[3];

void host_main(processor_state_struct *ps) {
    switch (((ps->pc - (1+(WORD)&labels[0]))) / sizeof(label_struct))) {
        case 0: ps->r1 = 9 << 2; /* tagged fixnum = 9 */
            ps->r0 = 1+(WORD)&labels[1]; /* tagged return address */
            ps->pc = glo_f; /* non-tail call f */
            return;
        case 1: printf("%d\n", ps->r1 >> 2);
            exit(0);
    }
}

void host_f(processor_state_struct *ps) {
    switch (((ps->pc - (1+(WORD)&labels[2]))) / sizeof(label_struct))) {
        case 0: ps->r1 += 4; /* add 1 to arg */
            ps->pc = glo_square; /* tail call square */
            return;
    }
}

void host_square(processor_state_struct *ps) {
    switch (((ps->pc - (1+(WORD)&labels[3]))) / sizeof(label_struct))) {
        case 0: ps->r1 *= (ps->r1 >> 2); /* square arg */
            ps->pc = ps->r0; /* return to main */
            return;
    }
}
Trampoline Optimizations

1. Locality is exploited by replacing `return` with a `goto` back to the `switch`

2. In *single host* compilation mode, all the code of a compiled file is generated in a single host function (exploits locality of jumps within the file)

3. Computed `goto`, available in gcc and clang, is faster than using a `switch`

4. Hosts can cache part of the processor state in local variables (improves likelihood C compiler will use machine registers)
Native Backend

• Uses machine registers to store the state of the GVM:

```plaintext
rdi    ;; r0
rax    ;; r1
rbx    ;; r2
rdx    ;; r3
rsi    ;; r4
rsp    ;; frame pointer
rbp    ;; heap pointer
rcx    ;; processor state pointer
```

• This assignment of machine registers is different than the one used by the C compiler (which we can’t and shouldn’t control because it would interfere with the optimizations of the C compiler)
Our Approach: Bridges and Dual Purpose label_struct

- Use handlers (bridges) for cross jumps that move the GVM state where the destination code expects it

- For Native backend: place executable machine code in label_structs starting at byte pointed by control point reference (to allow plain branch-to-address instructions in native code)

- Both backends use label_structs to represent control points:
  - Native backend: host field in label_struct points to the to_native_bridge
  - C backend: executable machine code in label_struct jumps to the from_native_bridge
**label_struct Layouts**

**x86-64 backend**

```c
void to_native_bridge(processor_state_struct *ps) { ... }
```

**C backend**

```c
void host(processor_state_struct *ps) { ... }
```

**C backend**

```c
from_native_bridge: mov [rcx], rdi ...
```
void to_native_bridge(processor_state_struct *ps) { ... }

void host(processor_state_struct *ps) { ... }

from_native_bridge: mov [rcx],rdi
...
**label_struct Layouts**

x86-64 backend

Reference to a control point

host  HEADER  entry_or_descr  machine code

```c
void to_native_bridge(processor_state_struct *ps) { ... }
```

Bridge called on C to Native cross jump

C backend

Reference to a control point

host  HEADER  entry_or_descr  code

```c
void host(processor_state_struct *ps) { ... }
```

Jumps to the machine code after storing the GVM state to registers

```c
from_native_bridge: mov [rcx],rdi
```
label_struct Layouts

x86-64 backend

Reference to a control point

void to_native_bridge(processor_state_struct *ps) { ... }

C backend

Reference to a control point

void host(processor_state_struct *ps) { ... }

Bridge called on Native to C cross jump

from_native_bridge: mov [rcx],rdi
... Saves GVM state from registers and jumps to host using trampoline
label_struct Layouts

void to_native_bridge(processor_state_struct *ps) { ... }

C to Native cross jump

from_native_bridge: mov [rcx],rdi ...

void host(processor_state_struct *ps) { ... }

C backend

x86-64 backend

Reference to a control point

host HEADER entry_or_descr

00 machine code

void to_native_bridge(processor_state_struct *ps) { ... }

C to Native cross jump

from_native_bridge: mov [rcx],rdi ...

void host(processor_state_struct *ps) { ... }

C backend

Reference to a control point

host HEADER entry_or_descr

00 machine code

x86-64 backend
x86-64 backend

Reference to a control point

host  HEADER  entry_or_descr

void to_native_bridge(processor_state_struct *ps) { ... }

C backend

Reference to a control point

host  HEADER  entry_or_descr  code

void host(processor_state_struct *ps) { ... }

from_native_bridge:  mov [rcx],rdi  ...

Note: host field moved to accommodate closures
entry field also needed for closures (see paper)
void to_native_bridge(processor_state_struct *ps) {
    __asm__ __volatile__ {
        "mov %0, %%rcx" // copy ps into %rcx
        "mov %rsp, -2*8(%%rcx)" // save C sp
        // setup handler for returning from native code
        "lea from_native_bridge(%%rip), %%rax"
        "mov %%rax, -1*8(%%rcx)" // setup handler
        // setup frame pointer and heap pointer registers
        "mov 5*8(%%rcx), %%rsp" // rsp = ps->fp
        "mov 6*8(%%rcx), %%rbp" // rbp = ps->hp
        // setup self register
        "mov 4*8(%%rcx), %%rsi" // rsi = ps->r4
        "mov 8*8(%%rcx), %%rax" // rax = ps->pc
        "cmpq $0x100000,-1-2*8(%%rax)" // closure?
        "jl setup_other_registers"
        "add $3, %%rsi" // handle closures
        "push %%rsi"
        "setup_other_registers:" // no outputs
        "mov (%%rcx), %%rdi" // rdi = ps->r0
        "mov 1*8(%%rcx), %%rax" // rax = ps->r1
        "mov 2*8(%%rcx), %%rbx" // rbx = ps->r2
        "mov 3*8(%%rcx), %%rdx" // rdx = ps->r3
        "jmp * 8*8(%%rcx)" // jump to ps->pc
        "from_native_bridge:" // inputs
        "mov %rdi, (%%rcx)" // ps->r0 = rdi
        "mov %rax, 1*8(%%rcx)" // ps->r1 = rax
        "mov %rbx, 2*8(%%rcx)" // ps->r2 = rbx
        "mov %rdx, 3*8(%%rcx)" // ps->r3 = rdx
    } // inputs
    // clobbers
    "%rdi","%rax","%rbx","%rdx","%rsi",
    "%rcx","%rbp";
};
}
Implementation of Bridges

```c
void to_native_bridge(processor_state_struct *ps) {
  __asm__
  __volatile__ {
    "mov %0, %%rcx"  // copy ps into %rcx
    "mov %rsp, -2*8(%%rcx)"  // save C sp
    // setup handler for returning from native code
    "lea from_native_bridge(%%rip), %%rax"  // handler
    "mov %%rax, -1*8(%%rcx)"  // setup handler
    // setup frame pointer and heap pointer registers
    "mov 5*8(%%rcx), %%rsp"  // rsp = ps->fp
    "mov 6*8(%%rcx), %%rbp"  // rbp = ps->hp
    // setup self register
    "mov 4*8(%%rcx), %%rsi"  // rsi = ps->r4
    "mov 8*8(%%rcx), %%rax"  // rax = ps->pc
    "cmpq $0x100000,-1-2*8(%%rax)"  // closure?
    "jl setup_other_registers"
    "add $3, %%rsi"  // handle closures
    "push %%rsi"
    "add $-3, %%rsi"
    "setup_other_registers:"  // no outputs
    "mov (%%rcx), %%rdi"     // rdi = ps->r0
    "mov 1*8(%%rcx), %%rax"  // rax = ps->r1
    "mov 2*8(%%rcx), %%rbx"  // rbx = ps->r2
    "mov 3*8(%%rcx), %%rdx"  // rdx = ps->r3
    "jmp * 8*8(%%rcx)" // jump to ps->pc
  }
}
```

```
to_native_bridge = 18 machine instructions
```

```c
from_native_bridge:
```

```
from_native_bridge = 15 machine instructions
```

```c
void from_native_bridge() {
    "mov %%rdi, (%%rcx)"  // ps->r0 = rdi
    "mov %%rax, 1*8(%%rcx)"  // ps->r1 = rax
    "mov %%rbx, 2*8(%%rcx)"  // ps->r2 = rbx
    "mov %%rdx, 3*8(%%rcx)"  // ps->r3 = rdx
    // recover destination control point in ps->pc
    "pop %%rax"
    "add $-3, %%rax"  // rax = destination ctrl pt
    "mov %%rax, 8*8(%%rcx)"  // ps->pc = rax
    "cmpq $0x100000,-1-2*8(%%rax)"  // closure?
    "jl store_self_register"
    "pop %%rsi"  // handle closures
    "add $-6, %%rsi"
    "store_self_register:"  // no inputs
    "mov %%rsi, 4*8(%%rcx)"  // ps->r4 = rsi
    "mov %%rsp, 5*8(%%rcx)"  // ps->fp = rsp
    "mov %%rbp, 6*8(%%rcx)"  // ps->hp = rbp
    "mov -2*8(%%rcx), %%rsp"  // restore C sp
}
```

```
from_native_bridge = 15 machine instructions
```
Implementation of Bridges

```c
void to_native_bridge(processor_state_struct *ps) {
    __asm__ volatile {
        "mov $0, %%rcx"  // copy ps into %rcx
        "mov %esp, -2*8(%%rcx)"  // save C sp
        // setup handler for returning from native code
        "lea from_native_bridge(%%rip), %%rax"  // setup handler
        "mov %%rax, -1*8(%%rcx)"  // setup handler
        // setup frame pointer and heap pointer registers
        "mov 5*8(%%rcx), %%rsp"  // rsp = ps->fp
        "mov 6*8(%%rcx), %%rbp"  // rbp = ps->hp
        // setup self register
        "mov 4*8(%%rcx), %%rsi"  // rsi = ps->r4
        "mov 8*8(%%rcx), %%rax"  // rax = ps->pc
        "add $3, %%rsi"  // handle closures
        "push %%rsi"
        "setup_other_registers:"  // setup other registers
        "mov (%%rcx), %%rdi"     // rdi = ps->r0
        "mov 1*8(%%rcx), %%rax"  // rax = ps->r1
        "mov 2*8(%%rcx), %%rbx"  // rbx = ps->r2
        "mov 3*8(%%rcx), %%rdx"  // rdx = ps->r3
        "jmp * 8*8(%%rcx)"  // jump to ps->pc
    }
}

from_native_bridge:
"mov %%rdi, (%%rcx)"  // ps->r0 = rdi
"mov %%rax, 1*8(%%rcx)"  // ps->r1 = rax
"mov %%rbx, 2*8(%%rcx)"  // ps->r2 = rbx
"mov %%rdx, 3*8(%%rcx)"  // ps->r3 = rdx
// recover destination control point in ps->pc
"pop %%rax"
"add $-3, %%rax"  // rax = destination ctrl pt
"mov %%rax, 8*8(%%rcx)"  // ps->pc = rax
"cmpq $0x100000,-1-2*8(%%rax)"  // closure?
"jl store_self_register"
"pop %%rsi"  // handle closures
"add $-6, %%rsi"
"store_self_register:"  // store self register
"mov %%rsi, 4*8(%%rcx)"  // ps->r4 = rsi
"mov %%rax, 5*8(%%rcx)"  // ps->r5 = rax
"mov %%rbp, 6*8(%%rcx)"  // ps->r6 = rbp
"mov -2*8(%%rcx), %%esp"  // restore C sp
}
```

```
// no outputs
// inputs
"m" (ps)
// clobbers
"%rdi", "%rax", "%rbx", "%rdx", "%rsi",
 "%rcx", "%rbp"
;
```

---

to_native_bridge = 18 machine instructions

from_native_bridge = 15 machine instructions

PLUS ONE ITERATION OF THE TRAMPOLINE

x86-64
Evaluation
Cost of Jumps

• Compile this Scheme code using C and Native backends and vary where `dec` procedure is defined (locally or in runtime system compiled by the C backend)

• 4 possible combinations exercise cross jumps and local jumps (purely in C code, or purely in Native code)

```
(declare (standard-bindings) (not safe))

(define (dec x) ;; variant of program defines (fx- x 1)) ;; dec in the runtime library

(define (run n f)
  (let loop ((n n))
    (if (fx> n 0)
      (if (fx> n 0)
        (loop (f n)))
      ;; call/return dec

  (time (run 100000000 dec))
```
## Cost of Jumps

<table>
<thead>
<tr>
<th></th>
<th>Native to Native</th>
<th>C to C (intra host)</th>
<th>C to C (inter host)</th>
<th>Native to C to Native</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x86-64 i7</strong></td>
<td>0.107 (0.07x)</td>
<td>0.238 (0.17x)</td>
<td>1.436 (1x)</td>
<td>2.264 (1.58x)</td>
</tr>
<tr>
<td><strong>x86-32 pIII</strong></td>
<td>2.947 (0.17x)</td>
<td>6.787 (0.40x)</td>
<td>16.854 (1x)</td>
<td>19.605 (1.16x)</td>
</tr>
<tr>
<td><strong>ARM</strong></td>
<td>2.758 (0.24x)</td>
<td>4.054 (0.35x)</td>
<td>11.431 (1x)</td>
<td>11.680 (1.02x)</td>
</tr>
</tbody>
</table>

Execution time in seconds and time relative to C backend with pure trampoline
# Cost of Jumps

<table>
<thead>
<tr>
<th></th>
<th>Native to Native</th>
<th>C to C (intra host)</th>
<th>C to C (inter host)</th>
<th>Native to C to Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64 i7</td>
<td>0.107 0.07x</td>
<td>0.238 0.17x</td>
<td>1.436 1x</td>
<td>2.264 1.58x</td>
</tr>
<tr>
<td>x86-32 pIII</td>
<td>2.947 0.17x</td>
<td>6.787 0.40x</td>
<td>16.854 1x</td>
<td>19.605 1.16x</td>
</tr>
<tr>
<td>ARM</td>
<td>2.758 0.24x</td>
<td>4.054 0.35x</td>
<td>11.431 1x</td>
<td>11.680 1.02x</td>
</tr>
</tbody>
</table>

Execution time in seconds and time relative to C backend with pure trampoline

- Cross jumps cost 2% to 58% more than pure trampoline
## Cost of Jumps

<table>
<thead>
<tr>
<th></th>
<th>Native to Native</th>
<th>C to C (intra host)</th>
<th>C to C (inter host)</th>
<th>Native to C to Native</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86-64 i7</td>
<td>0.107 0.07x</td>
<td>0.238 0.17x</td>
<td>1.436 1x</td>
<td>2.264 1.58x</td>
</tr>
<tr>
<td>x86-32 pIII</td>
<td>2.947 0.17x</td>
<td>6.787 0.40x</td>
<td>16.854 1x</td>
<td>19.605 1.16x</td>
</tr>
<tr>
<td>ARM</td>
<td>2.758 0.24x</td>
<td>4.054 0.35x</td>
<td>11.431 1x</td>
<td>11.680 1.02x</td>
</tr>
</tbody>
</table>

Execution time in seconds and time relative to C backend with pure trampoline

- Local jumps within code generated by Native backend are up to **14x** faster than inter host trampoline and **2.4x** faster than intra host trampoline
Conclusion
Desiderata

1. **Cross jumps are transparent** (All control points support both the trampoline and *branch-to-address*)

2. **Native code jumps are plain *branch-to-address instructions*** (The single machine jump instruction is optimal)

3. **Cost of cross jumps is similar to C backend local jumps** (Only 1.58x slower for x86-64, 1.16x for x86-32 and 1.02x for ARM)
Use cases

1. Useful tool for native backend development
   (all system features available throughout development)

2. Possibly completely avoid reimplementing the I/O and other complex subsystems
Motivation for Native Backend
Motivation for Native Backend

<table>
<thead>
<tr>
<th>Function</th>
<th>C Backend</th>
<th>x86-64 Backend</th>
<th>Chez Scheme 9.5.1 --optimize-level 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ack</td>
<td>10.943 1.13x</td>
<td>9.696 1x</td>
<td>12.392 1.28x</td>
</tr>
<tr>
<td>fib</td>
<td>2.989 1.19x</td>
<td>2.507 1x</td>
<td>4.249 1.69x</td>
</tr>
<tr>
<td>tak</td>
<td>7.628 0.98x</td>
<td>7.746 1x</td>
<td>8.135 1.05x</td>
</tr>
<tr>
<td>takl</td>
<td>3.993 1.02x</td>
<td>3.906 1x</td>
<td>7.576 1.94x</td>
</tr>
<tr>
<td>sum</td>
<td>2.878 0.77x</td>
<td>3.755 1x</td>
<td>11.784 3.14x</td>
</tr>
</tbody>
</table>

Unsafe fixnum benchmarks with frequent jumps
Gambit v4.9.0

(Preliminary results)