

GNU Jitter and the illusion of simplicity  
*or*  
Copying, patching and combining  
compiler-generated code in executable memory  
*or*  
The Anarchist's guide to GCC  
*or*  
The fun of playing with fire

Luca Saiu

<http://ageinghacker.net>

[positron@gnu.org](mailto:positron@gnu.org)

GNU Project

Binary T00ls Summit 2022

About these slides: Copyright © Luca Saiu 2022, released under the CC BY-SA 4.0 license.

The master copy is at <http://ageinghacker.net/talks/>



# Hello future viewers

The following resources are for those watching a recording of this presentation who want to occasionally pause the video and follow along:

- Source tarball for Structured, including GNU Jitter as a sub-package <http://ageinhacker.net/bts2022/structured-simple-1.0.tar.gz> (recommended)
- Source tarball for GNU Jitter, including Structured as a sub-package (yes, this is not a mistake), requiring bootstrap with the Autotools to compile Structured as in this demo: <http://ageinhacker.net/bts2022/jitter-0.9.285.tar.gz>
- Git source repository <http://git.ageinhacker.net/jitter>, requiring bootstrap with the Autotools plus Flex, GNU Bison, GNU Texinfo and so on. Use the tag `binary-tools-summit-2022` to get today's version.



# History and rationale

- Late 2016: I wanted to make GNU epsilon faster. Disappointed by threaded-code VMs;
- 2017: Read many scientific papers about making threaded-code VMs faster, mostly from [Anton Ertl](#) and the other [GForth](#) people (I recommend [Ertl and Gregg, 2004]); added my own ideas, generalised. Started Jitter.
- Presented Jitter at [GHM 2017](#); see the Talks page [Saiu, 2017] on my web site;
- Jitter used by GNU poke for its Poke Virtual Machine;
- December 2021: Jitter accepted as part of GNU.



# History and rationale

- Late 2016: I wanted to make GNU epsilon faster. Disappointed by threaded-code VMs;
- 2017: Read many scientific papers about making threaded-code VMs faster, mostly from [Anton Ertl](#) and the other [GForth](#) people (I recommend [Ertl and Gregg, 2004]); added my own ideas, generalised. Started Jitter.
- Presented Jitter at [GHM 2017](#); see the Talks page [Saiu, 2017] on my web site;
- Jitter used by GNU poke for its Poke Virtual Machine;
- December 2021: Jitter accepted as part of GNU.



# History and rationale

- Late 2016: I wanted to make GNU epsilon faster. Disappointed by threaded-code VMs;
- 2017: Read many scientific papers about making threaded-code VMs faster, mostly from [Anton Ertl](#) and the other [GForth](#) people (I recommend [Ertl and Gregg, 2004]); added my own ideas, generalised. Started Jitter.
- Presented Jitter at [GHM 2017](#); see the Talks page [Saiu, 2017] on my web site;
  - Jitter used by GNU poke for its Poke Virtual Machine;
  - December 2021: Jitter accepted as part of GNU.



# History and rationale

- Late 2016: I wanted to make GNU epsilon faster. Disappointed by threaded-code VMs;
- 2017: Read many scientific papers about making threaded-code VMs faster, mostly from [Anton Ertl](#) and the other [GForth](#) people (I recommend [Ertl and Gregg, 2004]); added my own ideas, generalised. Started Jitter.
- Presented Jitter at [GHM 2017](#); see the Talks page [Saiu, 2017] on my web site;
- Jitter [used by GNU poke](#) for its Poke Virtual Machine;
- December 2021: Jitter accepted as part of GNU.



# History and rationale

- Late 2016: I wanted to make GNU epsilon faster. Disappointed by threaded-code VMs;
- 2017: Read many scientific papers about making threaded-code VMs faster, mostly from [Anton Ertl](#) and the other [GForth](#) people (I recommend [Ertl and Gregg, 2004]); added my own ideas, generalised. Started Jitter.
- Presented Jitter at [GHM 2017](#); see the Talks page [Saiu, 2017] on my web site;
- Jitter [used by GNU poke](#) for its Poke Virtual Machine;
- December 2021: Jitter accepted as part of [GNU](#).



# Demo: the Structured programming language

## The Structured programming language

- Distributed along with Jitter as an example;
- Boring, unimaginative:
  - Pascal-style imperative language;
  - integer variables;
  - conditionals, loops;
  - recursive subprograms.
- Simple, clean implementation
  - Two VM code generators:
    - stack-based code;
    - register-based code(choose with a command-line-option);
  - minimal build system example including Jitter, with Autoconf / Automake;
- Fast with little effort

Demo





# Demo: the Structured programming language

## The Structured programming language

- Distributed along with Jitter as an example;
- **Boring**, unimaginative:
  - Pascal-style imperative language;
  - integer variables;
  - conditionals, loops;
  - recursive subprograms.
- Simple, clean implementation
  - Two VM code generators:
    - stack-based code;
    - register-based code(choose with a command-line-option);
  - minimal build system example including Jitter, with Autoconf / Automake;
- Fast with little effort

Demo



# Demo: the Structured programming language

## The Structured programming language

- Distributed along with Jitter as an example;
- **Boring**, unimaginative:
  - Pascal-style imperative language;
  - integer variables;
  - conditionals, loops;
  - recursive subprograms.
- **Simple, clean** implementation
  - **Two** VM code generators:
    - **stack-based** code;
    - **register-based** code(choose with a command-line-option);
  - minimal **build system example** including Jitter, with Autoconf / Automake;
- **Fast** with little effort

Demo



# Demo: the Structured programming language

## The Structured programming language

- Distributed along with Jitter as an example;
- **Boring**, unimaginative:
  - Pascal-style imperative language;
  - integer variables;
  - conditionals, loops;
  - recursive subprograms.
- **Simple, clean** implementation
  - **Two** VM code generators:
    - **stack-based** code;
    - **register-based** code(choose with a command-line-option);
  - minimal **build system example** including Jitter, with Autoconf / Automake;
- **Fast** with little effort

Demo



4/34

# Demo: the Structured programming language

The Structured programming language

- Distributed along with Jitter as an example;
- **Boring**, unimaginative:
  - Pascal-style imperative language;
  - integer variables;
  - conditionals, loops;
  - recursive subprograms.
- **Simple, clean** implementation
  - **Two** VM code generators:
    - **stack-based** code;
    - **register-based** code(choose with a command-line-option);
  - minimal **build system example** including Jitter, with Autoconf / Automake;
- **Fast** with little effort

## Demo



# Jitter contains some assembly

We are currently supporting ELF and COFF configurations using GCC on:

- [aarch64-unknown-linux-gnu](#)
- [alphaev4-unknown-linux-gnu](#)
- [alphaev67-unknown-linux-gnu](#)
- [arm-beaglebone-linux-gnueabi](#)
- [arm-replicant-linux-androideabi](#)  
(Android 6, tested on my Replicant. More recent versions probably work as well)
- [m68k-unknown-linux-gnu](#)
- [mipsel-unknown-linux-gnu](#)
- [mips-unknown-linux-gnu](#)
- [powerpcle-unknown-linux-gnu](#)
- [powerpc-unknown-linux-gnu](#)
- [riscv32-unknown-linux-gnu](#)
- [riscv64-unknown-linux-gnu](#)
- [s390x-ibm-linux-gnu](#)
- [sparc64-unknown-linux-gnu](#)
- [sparc-unknown-linux-gnu](#)
- [x86\\_64-unknown-haiku](#) (64-bit Haiku)
- [x86\\_64-unknown-linux-gnu](#)
- [x86\\_64-w64-mingw32](#)
- [x86\\_64-unknown-bsd\\*](#)

Not all of these configuration are supported as efficiently as you saw, yet.

More will come.



# Simple dispatches and why we are ignoring them

As an alternative to what you saw in the demo the **same** VMs can also run as **interpreters instead of JITs**.

One of two techniques:

- **switch** dispatch
- **direct-threading** dispatch

These are **slower** portability fallbacks, with **identical semantics**; [Saiu, 2017] shows how they work. **extremely portable**.

[If I have time: **minimal-threading** dispatch also exists as a middle-ground compromise between interpreter and JIT]

These alternatives exist but I will **ignore them** today. Today we deal with **GNU C** with **GCC** on a supported architecture with a supported binary format.



# Simple dispatches and why we are ignoring them

As an alternative to what you saw in the demo the **same** VMs can also run as **interpreters instead of JITs**.

One of two techniques:

- **switch** dispatch
- **direct-threading** dispatch

These are **slower** portability fallbacks, with **identical semantics**; [Saiu, 2017] shows how they work. **extremely portable**.

[If I have time: **minimal-threading** dispatch also exists as a middle-ground compromise between interpreter and JIT]

These alternatives exist but I will **ignore them** today. Today we deal with **GNU C** with **GCC** on a supported architecture with a supported binary format.



# Simple dispatches and why we are ignoring them

As an alternative to what you saw in the demo the **same** VMs can also run as **interpreters instead of JITs**.

One of two techniques:

- **switch** dispatch
- **direct-threading** dispatch

These are **slower** portability fallbacks, with **identical semantics**; [Saiu, 2017] shows how they work. **extremely portable**.

[If I have time: **minimal-threading** dispatch also exists as a middle-ground compromise between interpreter and JIT]

These alternatives exist but I will **ignore them** today. Today we deal with **GNU C** with **GCC** on a supported architecture with a supported binary format.





# Simple dispatches and why we are ignoring them

As an alternative to what you saw in the demo the **same** VMs can also run as **interpreters instead of JITs**.

One of two techniques:

- **switch** dispatch
- **direct-threading** dispatch

These are **slower** portability fallbacks, with **identical semantics**; [Saiu, 2017] shows how they work. **extremely portable**.

[If I have time: **minimal-threading** dispatch also exists as a middle-ground compromise between interpreter and JIT]

These alternatives exist but I will **ignore them** today. Today we deal with **GNU C** with **GCC** on a supported architecture with a supported binary format.



# Simple dispatches and why we are ignoring them

As an alternative to what you saw in the demo the **same** VMs can also run as **interpreters instead of JITs**.

One of two techniques:

- **switch** dispatch
- **direct-threading** dispatch

These are **slower** portability fallbacks, with **identical semantics**; [Saiu, 2017] shows how they work. **extremely portable**.

[If I have time: **minimal-threading** dispatch also exists as a middle-ground compromise between interpreter and JIT]

These alternatives exist but I will **ignore them** today. Today we deal with **GNU C** with **GCC** on a supported architecture with a supported binary format.



# Specialisation

Turn a generic VM instruction definition...

add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

...into every possible instantiation of register and immediate.

One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx; # Here %rdx is both input and output
add_r4_n1_r4_end:
```



# Specialisation

Turn a generic VM instruction definition...

add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

... into every possible instantiation of register and immediate.

One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx; # Here %rdx is both input and output
add_r4_n1_r4_end:
```



# Specialisation

Turn a generic VM instruction definition...

add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

... into every possible instantiation of register and immediate.

One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx # Here %rdx is both input and output
add_r4_n1_r4_end:
```



# Specialisation

Turn a generic VM instruction definition...

add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

... into every possible instantiation of register and immediate.

One example:

add specialisation r4/n1/r4: Generated C, macroexpanded, simplified

```
add_r4_n1_r4_begin:
  _local_state.r4 = _local_state.r4 + 1;
add_r4_n1_r4_end:
```

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
  addq $1, %rdx # Here %rdx is both input and output
add_r4_n1_r4_end:
```



# Replication

(The same block of hardware instructions shown above, delimited by two labels:)

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
    addq $1, %rdx
add_r4_n1_r4_end:
```

Easy: copy memory between the two labels into executable memory (allocated with `mmap`):

JIT-time replication: append an `add/r4/n1/r4` instruction

```
size_t vm_instruction_size_in_chars
    = ((char *) && add_r4_n1_r4_end
        - (char *) && add_r4_n1_r4_begin);
memcpy (executable_memory_end,
        && add_r4_n1_r4_begin,
        vm_instruction_size_in_chars);
executable_memory_end += vm_instruction_size_in_chars;
```



# Replication

(The same block of hardware instructions shown above, delimited by two labels:)

```
add specialisation r4/n1/r4, compiled
```

```
add_r4_n1_r4_begin:
    addq $1, %rdx
add_r4_n1_r4_end:
```

Easy: copy memory between the two labels into executable memory (allocated with `mmap`):

JIT-time replication: append an `add/r4/n1/r4` instruction

```
size_t vm_instruction_size_in_chars
    = ((char *) && add_r4_n1_r4_end
        - (char *) && add_r4_n1_r4_begin);
memcpy (executable_memory_end,
        && add_r4_n1_r4_begin,
        vm_instruction_size_in_chars);
executable_memory_end += vm_instruction_size_in_chars;
```





# Replication

(The same block of hardware instructions shown above, delimited by two labels:)

add specialisation r4/n1/r4, compiled

```
add_r4_n1_r4_begin:
    addq $1, %rdx
add_r4_n1_r4_end:
```

Easy: copy memory between the two labels into executable memory (allocated with `mmap`):

JIT-time replication: append an `add/r4/n1/r4` instruction

```
size_t vm_instruction_size_in_chars
    = ((char *) && add_r4_n1_r4_end
        - (char *) && add_r4_n1_r4_begin);
memcpy (executable_memory_end,
        && add_r4_n1_r4_begin,
        vm_instruction_size_in_chars);
executable_memory_end += vm_instruction_size_in_chars;
```



# Literal materialisation (1/2)

## add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

How do we specialise `add %r4, 3, %r4`?

- As `add_r4_nR_r4 ...`
  - `nR` means that the literal `number` is `Residual`: we `load into a register or memory via assembly code`, filled in by the Jitter runtime.

`add/_r4_nR_r4` compiled, with `0x3` as `nR`

```
movl $0x3,%r12d
movq %r12,%r8
```



# Literal materialisation (1/2)

## add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

How do we specialise `add %r4, 3, %r4`?

- As `add_r4_nR_r4 ...`  
`nR` means that the literal `number` is `Residual`: we **load into a register or memory via assembly code**, filled in by the Jitter runtime.

`add/_r4_nR_r4` compiled, with `3` as `nR`

```
movl $0x3,%r12d
movq %r12,%r8
```



# Literal materialisation (1/2)

## add VM instruction: Jitter specification

```
instruction add (?R, ?Rn 1, !R)
  code
    JITTER_ARGN2 = JITTER_ARGNO + JITTER_ARGN1;
  end
end
```

How do we specialise `add %r4, 3, %r4`?

- As `add_r4_nR_r4 ...`  
`nR` means that the literal `number` is `Residual`: we **load into a register or memory via assembly code**, filled in by the Jitter runtime.

## add/\_r4\_nR\_r4 compiled, with `0x3` as `nR`

```
movl $0x3,%r12d
movq %r12,%r8
```



## Literal materialisation (2/2)

We can even have different assembly code for different constants. For example on x86\_64 the literal 0x0 can be loaded in a more efficient way than other numbers.

```
add/_r4_nR_r4 compiled, with 0x0 as r4
xorl %r12d,%r12d # This only works for 0x0
movq %r12,%r8
```

Same for small constants on most RISCs.

(A hardware register has been reserved)

This is very architecture-specific, and a little laborious, but...

...VM-independent: the complexity is all in Jitter!



## Literal materialisation (2/2)

We can even have different assembly code for different constants. For example on x86\_64 the literal 0x0 can be loaded in a more efficient way than other numbers.

add/\_r4\_nR\_r4 compiled, with 0x0 as nR

```
xorl %r12d,%r12d # This only works for 0x0  
movq %r12,%r8
```

Same for small constants on most RISCs.

(A hardware register has been reserved)

This is very architecture-specific, and a little laborious, but...

...VM-independent: the complexity is all in Jitter!



## Literal materialisation (2/2)

We can even have different assembly code for different constants. For example on x86\_64 the literal 0x0 can be loaded in a more efficient way than other numbers.

add/\_r4\_nR\_r4 compiled, with 0x0 as nR

```
xorl %r12d,%r12d # This only works for 0x0
movq %r12,%r8
```

Same for small constants on most RISCs.

(A hardware register has been reserved)

This is very architecture-specific, and a little laborious, but...

...VM-independent: the complexity is all in Jitter!



## Literal materialisation (2/2)

We can even have different assembly code for different constants. For example on x86\_64 the literal 0x0 can be loaded in a more efficient way than other numbers.

add/\_r4\_nR\_r4 compiled, with 0x0 as nR

```
xorl %r12d,%r12d # This only works for 0x0
movq %r12,%r8
```

Same for small constants on most RISCs.

(A hardware register has been reserved)

This is very architecture-specific, and a little laborious, but...

...VM-independent: the complexity is all in Jitter!





## Literal materialisation (2/2)

We can even have different assembly code for different constants. For example on x86\_64 the literal 0x0 can be loaded in a more efficient way than other numbers.

add/\_r4\_nR\_r4 compiled, with 0x0 as nR

```
xorl %r12d,%r12d # This only works for 0x0  
movq %r12,%r8
```

Same for small constants on most RISCs.

(A hardware register has been reserved)

This is very architecture-specific, and a little laborious, but...

...VM-independent: the complexity is all in Jitter!



## Literal materialisation (2/2)

We can even have different assembly code for different constants. For example on x86\_64 the literal 0x0 can be loaded in a more efficient way than other numbers.

add/\_r4\_nR\_r4 compiled, with 0x0 as nR

```
xorl %r12d,%r12d # This only works for 0x0
movq %r12,%r8
```

Same for small constants on most RISCs.

(A hardware register has been reserved)

This is very architecture-specific, and a little laborious, but...

... VM-independent: **the complexity is all in Jitter!**



# Fast branches (1/1)

How do we perform jumps from a VM instruction to another?

- One way would be by materialisation:

That works (I used to do that) but is **slow**.

- Much better solution: generate something like

```
b $f compiled, with $f being patched in  
  jmp .L # .L will be overwritten after replication
```



# Fast branches (1/1)

How do we perform jumps from a VM instruction to another?

- One way would be by materialisation:
  - load the target address into a register
  - jump via register

That works (I used to do that) but is **slow**.

- Much better solution: generate something like

```
b $f compiled, with $f being patched in  
jmp .L # .L will be overwritten after replication
```



# Fast branches (1/1)

How do we perform jumps from a VM instruction to another?

- One way would be by materialisation:
  - load the target address into a register
  - jump via register

That works (I used to do that) but is **slow**.

- Much better solution: generate something like

```
b $f compiled, with $f being patched in  
jmp .L # .L will be overwritten after replication
```



# Fast branches (1/1)

How do we perform jumps from a VM instruction to another?

- One way would be by materialisation:
  - load the target address into a register
  - jump via register

That works (I used to do that) but is **slow**.

- Much better solution: generate something like

```
b $f compiled, with $f being patched in
```

```
jmp .L # .L will be overwritten after replication
```



# Fast branches (2/2)

There are two main problems

- How to present this to the user in a friendly way

- `JITTER_BRANCH_FAST(label)`,
  - `JITTER_BRANCH_FAST_IF_NONZERO(expression, label)`
  - `JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW(lvalue, rvalue, rvalue, label) ...`

- How to implement this (**where to patch**)?

- We need a new idea: **patch-ins**.

[Quick demo with uninspired

[unless I am very late]

- conditional branch

- plus-or-branch-if-overflow]



# Fast branches (2/2)

There are two main problems

- How to present this to the user in a friendly way
  - `JITTER_BRANCH_FAST(label)`,  
`JITTER_BRANCH_FAST_IF_NONZERO(expression, label)`  
`JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW(lvalue,  
rvalue, rvalue, label) ...`
- How to implement this (**where to patch**)?
  - We need a new idea: **patch-ins**.

[Quick demo with uninspired

[unless I am very late]

- conditional branch

- plus-or-branch-if-overflow]





# Fast branches (2/2)

There are two main problems

- How to present this to the user in a friendly way
  - `JITTER_BRANCH_FAST(label)`,  
`JITTER_BRANCH_FAST_IF_NONZERO(expression, label)`  
`JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW(lvalue,  
rvalue, rvalue, label) ...`
- How to implement this (**where to patch**)?
  - We need a new idea: **patch-ins**.

[Quick demo with uninspired

[unless I am very late]

- conditional branch

- plus-or-branch-if-overflow]



# Fast branches (2/2)

There are two main problems

- How to present this to the user in a friendly way
  - `JITTER_BRANCH_FAST(label)`,  
`JITTER_BRANCH_FAST_IF_NONZERO(expression, label)`  
`JITTER_PLUS_BRANCH_FAST_IF_OVERFLOW(lvalue,  
rvalue, rvalue, label) ...`
- How to implement this (**where to patch**)?
  - We need a new idea: **patch-ins**.

[Quick demo with uninspired

[unless I am very late]

- conditional branch
- plus-or-branch-if-overflow]



# Patch-ins

The fast-branching macro expands to something like:

(Macro-expanded) GNU C, simplified

```
asm goto (".pushsection .data, 42\n"
         "   .quad hole_to_fill_%=\n"
         "   .quad " SPECIALIZED_INSTRUCTION_ID " \n"
         "   .quad " PATCH_IN_CASE " \n"
         ".popsection\n"
         "hole_to_fill_%=: \n"
         "   .skip " ROUTINE_LENGTH_IN_BYTES " \n"
         " : /* inputs... */
         " : unreachable_label_jumping_where_gcc_cant_know);
```

This trick uses subsections. A pointer to the memory to patch is stored in a global table; the displacement between that address and an instruction start address is a literal constant.



# C restrictions: global variables

C globals are accessed with **PC-relative** instructions on modern architectures.

- `a_global`  
 $\implies$  `(* _a_local_struct->a_global_address)`
- wrapping mechanism:  

```
#define a_global \
    (* _a_local_struct->a_global_address)
```

## Jitter specification

```
wrapped-globals
  my_datum
  prompt_string
end
```



# C restrictions: global variables

C globals are accessed with **PC-relative** instructions on modern architectures.

- `a_global`  
 $\implies$  `(* _a_local_struct->a_global_address)`
- wrapping mechanism:  

```
#define a_global \
    (* _a_local_struct->a_global_address)
```

## Jitter specification

```
wrapped-globals
  my_datum
  prompt_string
end
```



# C restrictions: global variables

C globals are accessed with **PC-relative** instructions on modern architectures.

- `a_global`  
 $\implies (*\_a\_local\_struct \rightarrow a\_global\_address)$
- wrapping mechanism:  

```
#define a_global \
    (* _a_local_struct->a_global_address)
```

## Jitter specification

```
wrapped-globals
  my_datum
  prompt_string
end
```



# C restrictions: calling C functions (1/3)

Functions are also accessed with **PC-relative** instructions.

Same trick necessary...

- `a_function`  
⇒ `(* _a_local_struct->a_function_address)`

Jitter specification

```
wrapped-functions
```

```
  putc
```

```
  getc
```

```
end
```

... but this is **not enough!**



# C restrictions: calling C functions (1/3)

Functions are also accessed with **PC-relative** instructions.  
Same trick necessary. . .

- `a_function`  
 $\implies (*\_a\_local\_struct \rightarrow a\_function\_address)$
- wrapping mechanism:  

```
#define a_function \
    (* _a_local_struct->a_function_address)
```

Jitter specification

wrapped-functions

```
putc
getc
end
```

... but this is **not enough!**





# C restrictions: calling C functions (1/3)

Functions are also accessed with **PC-relative** instructions.  
Same trick necessary. . .

- `a_function`  
 $\implies$  `(* _a_local_struct->a_function_address)`
- wrapping mechanism:  

```
#define a_function \
    (* _a_local_struct->a_function_address)
```

Jitter specification

wrapped-functions

```
putc
getc
end
```

... but this is **not enough!**



# C restrictions: calling C functions (1/3)

Functions are also accessed with **PC-relative** instructions.  
Same trick necessary. . .

- `a_function`  
 $\implies (*\_a\_local\_struct \rightarrow a\_function\_address)$
- wrapping mechanism:  

```
#define a_function \
    (* _a_local_struct->a_function_address)
```

## Jitter specification

```
wrapped-functions
  putc
  getc
end
```

... but this is **not enough!**



# C restrictions: calling C functions (1/3)

Functions are also accessed with **PC-relative** instructions.  
Same trick necessary. . .

- `a_function`  
 $\implies (*\_a\_local\_struct \rightarrow a\_function\_address)$
- wrapping mechanism:  

```
#define a_function \
    (* _a_local_struct->a_function_address)
```

## Jitter specification

```
wrapped-functions
  putc
  getc
end
```

. . . but this is **not enough!**



# C restrictions: calling C functions (2/3)

On some architectures / ABIs **calling a C function clobbers CPU state**: for example the global pointer register on Alpha, or a floating point status register on SH4. [If I have time: violating the C ABI with respect to call-clobbered registers can be useful]

- A wrapped function call should expand to an expression “like”:

```
{  
  PRE_PROCEDURE_CALL_CODE;  
  int _res  
  = (* _a_local_struct->a_function_address) (args);  
  POST_PROCEDURE_CALL_CODE;  
  _res; }  
}
```



## C restrictions: calling C functions (2/3)

On some architectures / ABIs **calling a C function clobbers CPU state**: for example the global pointer register on Alpha, or a floating point status register on SH4. [If I have time: violating the C ABI with respect to call-clobbered registers can be useful]

- A wrapped function call should expand to an expression “like”:

```
({ PRE_PROCEDURE_CALL_CODE;  
  int _res ← any output type, not just int  
    = (* _a_local_struct->a_function_address) (args);  
  POST_PROCEDURE_CALL_CODE;  
  _res; })
```
- The macro must work with any arity, any input type, any output type — and void is special.
- The actual definition is complex and requires GNU C (but is well factored).



## C restrictions: calling C functions (2/3)

On some architectures / ABIs **calling a C function clobbers CPU state**: for example the global pointer register on Alpha, or a floating point status register on SH4. [If I have time: violating the C ABI with respect to call-clobbered registers can be useful]

- A wrapped function call should expand to an expression “like”:  
(`{ PRE_PROCEDURE_CALL_CODE;`  
    `int _res`  $\leftarrow$  **any output type, not just int**  
    `= (* _a_local_struct->a_function_address) (args);`  
    `POST_PROCEDURE_CALL_CODE;`  
    `_res; }`)
- The macro must work with any arity, any input type, **any output type** — and void is special.
- The actual definition is **complex** and requires **GNU C** (but is well factored).



# C restrictions: calling C functions (2/3)

On some architectures / ABIs **calling a C function clobbers CPU state**: for example the global pointer register on Alpha, or a floating point status register on SH4. [If I have time: violating the C ABI with respect to call-clobbered registers can be useful]

- A wrapped function call should expand to an expression “like”:  
(`{ PRE_PROCEDURE_CALL_CODE;`  
    `int _res`  $\leftarrow$  **any output type, not just int**  
    `= (* _a_local_struct->a_function_address) (args);`  
    `POST_PROCEDURE_CALL_CODE;`  
    `_res; }`)
- The macro must work with any arity, any input type, **any output type** — and void is special.
- The actual definition is **complex** and requires **GNU C** (but is well factored).



# C restrictions: calling C functions (3/3)

- The actual macro needs:
  - statement-expressions;
  - `typeof`;
  - `__builtin_choose_expr`;
  - `__builtin_types_compatible_p`;
  - different definition for functions returning void.

For more information use `git grep 'Function wrapping'`

- Still very easy to use:

```
Jitter specification
```

```
wrapped-functions
```

```
  putc
```

```
  getc
```

```
end
```





# C restrictions: calling C functions (3/3)

- The actual macro needs:
  - statement-expressions;
  - `typeof`;
  - `__builtin_choose_expr`;
  - `__builtin_types_compatible_p`;
  - different definition for functions returning void.

For more information use `git grep 'Function wrapping'`

- Still `very easy to use`:

## Jitter specification

```
wrapped-functions
  putc
  getc
end
```



# C operators implemented as out-of-line routines

Very architecture-dependent:

- Sometimes solvable by command-line options.

Ex.: struct assignment on PowerPC:

```
-mblock-move-inline-limit=8192
```

```
-fcommon-point-headers
```

```
-fcommon
```



# C operators implemented as out-of-line routines

Very architecture-dependent:

- Sometimes solvable by command-line options.

Ex.: struct assignment on PowerPC:

```
-mblock-move-inline-limit=8192
```

- floating-point literals;
- division;



# C operators implemented as out-of-line routines

Very architecture-dependent:

- Sometimes solvable by command-line options.  
Ex.: struct assignment on PowerPC:  
`-mblock-move-inline-limit=8192`
- floating-point literals;
- division;



# C operators implemented as out-of-line routines

Very architecture-dependent:

- Sometimes solvable by command-line options.  
Ex.: struct assignment on PowerPC:  
`-mblock-move-inline-limit=8192`
- floating-point literals;
- division;
  - automatically wrap libgcc internal functions; it seems to work (!) — GCC experts, do you know why it works? I have no idea
  - dividing VM instruction attribute — easy, but not implemented yet



# C operators implemented as out-of-line routines

Very architecture-dependent:

- Sometimes solvable by command-line options.  
Ex.: struct assignment on PowerPC:  
`-mblock-move-inline-limit=8192`
- floating-point literals;
- division;
  - automatically `wrap` libgcc internal functions; it seems to work (!) — GCC experts, do you know **why it works?** I have no idea
  - `dividing` VM instruction attribute — easy, but not implemented yet



# C operators implemented as out-of-line routines

Very architecture-dependent:

- Sometimes solvable by command-line options.  
Ex.: struct assignment on PowerPC:  
`-mblock-move-inline-limit=8192`
- floating-point literals;
- division;
  - automatically `wrap` libgcc internal functions; it seems to work (!) — GCC experts, do you know **why it works?** I have no idea
  - `dividing` VM instruction attribute — easy, but not implemented yet



# C restrictions: summary

What we have learned up to this point:

- VM instruction **code blocks require care**:
  - write C...
  - ... think assembly
- No restrictions in called C functions;
- No restrictions in **non-relocatable** VM instructions;
- Relatively minor annoyances.





# C restrictions: summary

What we have learned up to this point:

- VM instruction **code blocks require care**:
  - **write C...**
  - **...think assembly**
- No restrictions in **called C functions**;
- No restrictions in **non-relocatable VM instructions**;
- **Relatively minor annoyances**.



# C restrictions: summary

What we have learned up to this point:

- VM instruction **code blocks require care**:
  - **write C...**
  - ... **think assembly**
- No restrictions in **called C functions**;
- No restrictions in **non-relocatable** VM instructions;
- **Relatively minor annoyances.**



# C restrictions: summary

What we have learned up to this point:

- VM instruction **code blocks require care**:
  - **write C...**
  - **... think assembly**
- No restrictions in **called C functions**;
- No restrictions in **non-relocatable** VM instructions;
- **Relatively minor annoyances.**



# C restrictions: summary

What we have learned up to this point:

- VM instruction **code blocks require care**:
  - **write C...**
  - **... think assembly**
- No restrictions in **called C functions**;
- No restrictions in **non-relocatable** VM instructions;
- **Relatively minor** annoyances.



# What can *really* go wrong

GCC can compile C in ways that are **legitimate**, but **break our strategy** of copying and recombining hardware instruction blocks.

- reorder;
- tail-merging;
- inconsistent register assignments across branches;
- far branches;
- PC-relative loads to materialise constants.



# What can go wrong: reorder (1/2: the problem)

When replicating we memcopy from `the_beginning` to `the_end`:

A specialised instruction, compiled

```
the_beginning:
    ...
    ...
the_end:
```

But GCC might (legitimately) `reorder blocks`:

A specialised instruction, compiled — negative length?

```
.foo:
    ...
the_end:
    ... # Something else
the_beginning:
    ...
    jmp .foo
```

Code from `different VM instructions` may even be `intertwined`.



# What can go wrong: reorder (1/2: the problem)

When replicating we memcopy from `the_beginning` to `the_end`:

A specialised instruction, compiled

```
the_beginning:
...
...
the_end:
```

But GCC might (legitimately) **reorder blocks**:

A specialised instruction, compiled — negative length?

```
.foo:
...
the_end:
... # Something else
the_beginning:
...
jmp .foo
```

Code from **different VM instructions** may even be **intertwined**.



# What can go wrong: reorder (1/2: the problem)

When replicating we memcpy from `the_beginning` to `the_end`:

A specialised instruction, compiled

```
the_beginning:
...
...
the_end:
```

But GCC might (legitimately) **reorder blocks**:

A specialised instruction, compiled — negative length?

```
.foo:
...
the_end:
... # Something else
the_beginning:
...
jmp .foo
```

Code from **different** VM instructions may even be **intertwined**.





## What can go wrong: reorder (2/2: the solution)

In this case the solution is easy: GCC has a `-fno-reorder-blocks` option

The generated file `vmprefix-vm2.c` **must** be compiled with `-fno-reorder-blocks`.

This GCC optimisation option is **necessary for correctness!**

`JITTER_CFLAGS` contains it.

(Out of defensiveness, VM code checks at startup that specialised instruction blocks are disjoint and sizes are non-negative.)



## What can go wrong: reorder (2/2: the solution)

In this case the solution is easy: GCC has a `-fno-reorder-blocks` option

The generated file `vmprefix-vm2.c` **must** be compiled with `-fno-reorder-blocks`.

This GCC optimisation option is **necessary for correctness!**

`JITTER_CFLAGS` contains it.

(Out of defensiveness, VM code checks at startup that specialised instruction blocks are disjoint and sizes are non-negative.)



## What can go wrong: reorder (2/2: the solution)

In this case the solution is easy: GCC has a `-fno-reorder-blocks` option

The generated file `vmprefix-vm2.c` **must** be compiled with `-fno-reorder-blocks`.

This GCC optimisation option is **necessary for correctness!**

`JITTER_CFLAGS` contains it.

(Out of defensiveness, VM code checks at startup that specialised instruction blocks are disjoint and sizes are non-negative.)



## What can go wrong: reorder (2/2: the solution)

In this case the solution is easy: GCC has a `-fno-reorder-blocks` option

The generated file `vmprefix-vm2.c` **must** be compiled with `-fno-reorder-blocks`.

This GCC optimisation option is **necessary for correctness!**

`JITTER_CFLAGS` contains it.

(Out of defensiveness, VM code checks at startup that specialised instruction blocks are disjoint and sizes are non-negative.)



## What can go wrong: reorder (2/2: the solution)

In this case the solution is easy: GCC has a `-fno-reorder-blocks` option

The generated file `vmprefix-vm2.c` **must** be compiled with `-fno-reorder-blocks`.

This GCC optimisation option is **necessary for correctness!**

`JITTER_CFLAGS` contains it.

(Out of defensiveness, VM code checks at startup that specialised instruction blocks are disjoint and sizes are non-negative.)



# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

```
foo VM instruction
foo_beginning:
...
xxx
yyy
foo_end:
```

```
bar VM instruction
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar factoring their common tail:

```
foo VM instruction
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

```
bar VM instruction
bar_beginning:
...
jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

**Crash.**



# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

## foo VM instruction

```
foo_beginning:
...
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar **factoring their common tail**:

## foo VM instruction

```
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
  jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

**Crash.**



# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

```
foo VM instruction
foo_beginning:
...
xxx
yyy
foo_end:
```

```
bar VM instruction
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar factoring their common tail:

```
foo VM instruction
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

```
bar VM instruction
bar_beginning:
...
jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

**Crash.**





# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

## foo VM instruction

```
foo_beginning:
...
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar factoring their common tail:

## foo VM instruction

```
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

**Crash.**



# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

## foo VM instruction

```
foo_beginning:
...
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar factoring their common tail:

## foo VM instruction

```
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

Crash.



# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

## foo VM instruction

```
foo_beginning:
...
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar factoring their common tail:

## foo VM instruction

```
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

## bar VM instruction

```
bar_beginning:
...
jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

**Crash.** (If you are lucky.)



# What can go wrong: tail merging (1/3: the problem)

Two VM instructions happen to behave the same way at the end:

```
foo VM instruction
foo_beginning:
...
xxx
yyy
foo_end:
```

```
bar VM instruction
bar_beginning:
...
xxx
yyy
bar_end:
```

Or GCC might (legitimately) compile foo and bar factoring their common tail:

```
foo VM instruction
foo_beginning:
...
.tail_of_foo:
xxx
yyy
foo_end:
```

```
bar VM instruction
bar_beginning:
...
jmp .tail_of_foo # Hmm...
bar_end: # If at all
```

After copying, at run time, the jump will reach **out of** the relocated bar\_beginning and bar\_end.

**Crash. (If you are lucky.)**



# What can go wrong: tail merging (2/3: not enough)

GCC has a `-fno-tail-merge` option...

...which is, unfortunately, **not enough**: GCC merges some tails even with the option. [Changing GCC now would not be very useful in practice: many stable releases with this behaviour are in use]

We need a more creative solution.



# What can go wrong: tail merging (2/3: not enough)

GCC has a `-fno-tail-merge` option...

... which is, unfortunately, **not enough**: GCC merges some tails even with the option. [Changing GCC now would not be very useful in practice: many stable releases with this behaviour are in use]

We need a more creative solution.



# What can go wrong: tail merging (2/3: not enough)

GCC has a `-fno-tail-merge` option...

... which is, unfortunately, **not enough**: GCC merges some tails even with the option. [Changing GCC now would not be very useful in practice: many stable releases with this behaviour are in use]

We need a more creative solution.



# What can go wrong: tail merging (3/3: the solution)

Use **GNU C assembly constraints** to pretend there are dependencies, and that a variable is modified in many different ways: **prevent factoring by making tails appear different.**

foo VM instruction, in C

```
foo_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 123");
foo_end:
    asm (: "+X" (variable),
        "# Useless 124");
    goto * variable;
```

bar VM instruction, in C

```
bar_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 125");
bar_end:
    asm (: "+X" (variable),
        "# Useless 126");
    goto * variable;
```

The variable must actually be used.

[The Array's base is in a reserved register: the obvious candidate, but I am not speaking of The Array in this presentation]

A lot of Jitter code is based on this principle: **lie to the compiler!**





# What can go wrong: tail merging (3/3: the solution)

Use **GNU C assembly constraints** to pretend there are dependencies, and that a variable is modified in many different ways: **prevent factoring by making tails appear different.**

foo VM instruction, in C

```
foo_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 123");
foo_end:
    asm (: "+X" (variable),
        "# Useless 124");
    goto * variable;
```

bar VM instruction, in C

```
bar_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 125");
bar_end:
    asm (: "+X" (variable),
        "# Useless 126");
    goto * variable;
```

The variable must actually be used.

[The Array's base is in a reserved register: the obvious candidate, but I am not speaking of The Array in this presentation]

A lot of Jitter code is based on this principle: **lie to the compiler!**



# What can go wrong: tail merging (3/3: the solution)

Use **GNU C assembly constraints** to pretend there are dependencies, and that a variable is modified in many different ways: **prevent factoring by making tails appear different**.

foo VM instruction, in C

```
foo_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 123");
foo_end:
    asm (: "+X" (variable),
        "# Useless 124");
    goto * variable;
```

bar VM instruction, in C

```
bar_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 125");
bar_end:
    asm (: "+X" (variable),
        "# Useless 126");
    goto * variable;
```

The variable must actually be used.

[The Array's base is in a reserved register: the obvious candidate, but I am not speaking of The Array in this presentation]

A lot of Jitter code is based on this principle: **lie to the compiler!**



# What can go wrong: tail merging (3/3: the solution)

Use **GNU C assembly constraints** to pretend there are dependencies, and that a variable is modified in many different ways: **prevent factoring by making tails appear different.**

foo VM instruction, in C

```
foo_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 123");
foo_end:
    asm (: "+X" (variable),
        "# Useless 124");
    goto * variable;
```

bar VM instruction, in C

```
bar_beginning:
    ...
    asm (: "+X" (variable),
        "# Useless 125");
bar_end:
    asm (: "+X" (variable),
        "# Useless 126");
    goto * variable;
```

The variable must actually be used.

[The Array's base is in a reserved register: the obvious candidate, but I am not speaking of The Array in this presentation]

A lot of Jitter code is based on this principle: **lie to the compiler!**



# Register-assignments across branches

Can we guarantee that GCC uses the same register assignment at a branch site and at the target site?

- the target is always the beginning of a VM instruction

compiled code going to `Ltarget`

```

    jmp .Ltemp
    ...
.Ltemp:
    mov ..., %rdx # Register assignment now compatible with Ltarget
    jmp Ltarget

```

[To do: There will be no time for the details: the solution uses `asm goto` in a creative way, lying to the compiler]



# Register-assignments across branches

Can we guarantee that GCC uses the same register assignment at a branch site and at the target site?

- the target is always the **beginning of a VM instruction**

compiled code going to `Ltarget`

```

    jmp .Ltemp
    ...
.Ltemp:
    mov ..., %rdx # Register assignment now compatible with Ltarget
    jmp Ltarget

```

[To do: There will be no time for the details: the solution uses `asm goto` in a creative way, lying to the compiler]



# Register-assignments across branches

Can we guarantee that GCC uses the same register assignment at a branch site and at the target site?

- the target is always the **beginning of a VM instruction**

compiled code going to **Ltarget**

GCC can (legitimately) generate code like this:

```

    jmp .Ltemp
    ...
.Ltemp:
    mov ..., %rdx # Register assignment now compatible with Ltarget
    jmp Ltarget

```

[To do: There will be no time for the details: the solution uses `asm goto` in a creative way, lying to the compiler]



# Register-assignments across branches

Can we guarantee that GCC uses the same register assignment at a branch site and at the target site?

- the target is always the **beginning of a VM instruction**

compiled code going to **Ltarget**

GCC can (legitimately) generate code like this:

```

    jmp .Ltemp
    ...
.Ltemp:
    mov ..., %rdx # Register assignment now compatible with Ltarget
    jmp Ltarget

```

**[To do:** There will be no time for the details: the solution uses **asm goto** in a creative way, lying to the compiler]



# Far branches

[**To do:** There will be no time for this]





# PC-relative loads to materialise constants

[**To do:** There will be no time for this]



# Code reuse

- Good architecture-dependent support is **complicated**
  - Possibly **unjustified effort** for a single VM
  - It will not happen unless the author enjoys assembly and low-level programming
  - Architecture-dependent, VM-independent
  - Jitter is the **right way to factor**
- the code needed to play with fire



# Code reuse

- Good architecture-dependent support is **complicated**
- Possibly **unjustified effort** for a single VM
- It will not happen unless the author enjoys assembly and low-level programming
  - (Notice that **I do**)
- Architecture-dependent, VM-independent
- Jitter is the **right way to factor**



# Code reuse

- Good architecture-dependent support is **complicated**
- Possibly **unjustified effort** for a single VM
- It will not happen unless the author enjoys assembly and low-level programming
  - (Notice that **I do**)
- Architecture-dependent, VM-independent
- Jitter is the **right way to factor**



# Code reuse

- Good architecture-dependent support is **complicated**
- Possibly **unjustified effort** for a single VM
- It will not happen unless the author enjoys assembly and low-level programming
  - (Notice that **I do**)
- **Architecture-dependent**, **VM-independent**
- Jitter is the **right way to factor**



# Code reuse

- Good architecture-dependent support is **complicated**
- Possibly **unjustified effort** for a single VM
- It will not happen unless the author enjoys assembly and low-level programming
  - (Notice that **I do**)
- **Architecture-dependent**, **VM-independent**
- Jitter is the **right way to factor**
  - Jitter users do **not** need to play with fire
  - Jittery VMs are **high-level**



# Code reuse

- Good architecture-dependent support is **complicated**
- Possibly **unjustified effort** for a single VM
- It will not happen unless the author enjoys assembly and low-level programming
  - (Notice that **I do**)
- **Architecture-dependent**, **VM-independent**
- Jitter is the **right way to factor**
  - Jitter users do **not** need to play with fire
  - Jittery VMs are **high-level**



# Code reuse

- Good architecture-dependent support is **complicated**
- Possibly **unjustified effort** for a single VM
- It will not happen unless the author enjoys assembly and low-level programming
  - (Notice that **I do**)
- **Architecture-dependent**, **VM-independent**
- Jitter is the **right way to factor**
  - Jitter users do **not** need to play with fire
  - Jittery VMs are **high-level**





# Is this the right thing?

## Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;
  - The VM's complexity comes from the "real flexibility" of the host language, which is necessary to build a C compiler without paying the performance cost of using performance.

## Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.
  - We should be prepared to accept some compromises.



# Is this the right thing?

Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;

Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.

• We should be prepared to accept



# Is this the right thing?

Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;
  - Jitter's complexity comes from the lack of flexibility of C;
  - Impossible to reuse a C compiler without playing with fire, if we want to keep performance.

Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.



# Is this the right thing?

Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;
  - Jitter's complexity comes from the lack of flexibility of C;
  - Impossible to reuse a C compiler without playing with fire, if we want to keep performance.

Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.



# Is this the right thing?

Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;
  - Jitter's complexity comes from the lack of flexibility of C;
  - Impossible to reuse a C compiler without playing with fire, if we want to keep performance.

Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.
- We should be prepared to restart from scratch



# Is this the right thing?

Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;
  - Jitter's complexity comes from the lack of flexibility of C;
  - Impossible to reuse a C compiler without playing with fire, if we want to keep performance.

Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.
- We should be prepared to **restart from scratch**
  - Factor. Simplify. **Reject compatibility.**



# Is this the right thing?

Is this complexity necessary?

- In an ideal world a VM generator would be simpler.
- Jitter is a product of its environment;
  - Jitter's complexity comes from the lack of flexibility of C;
  - Impossible to reuse a C compiler without playing with fire, if we want to keep performance.

Is the environment complexity necessary?

- Unix is too complex;
- C is too complex.
- We should be prepared to **restart from scratch**
  - Factor. Simplify. **Reject compatibility.**



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation (⇒ Info, PDF, HTML, ...):
  - [The Jitter manual](#)
  - [The example manual](#)
  - [The manual manual](#)
  - [The programmer manual](#)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`





# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual
  - Jitter example manuals:
    - Structured manual
    - JitterLisp manual
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (*very, very incomplete*)
    - Jitter example manuals:
      - Structured manual
      - JitterLisp manual
  - Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
    - Jitter example manuals:
      - Structured manual
      - JitterLisp manual
  - Tutorial at <http://ageinghacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (*existing as an empty stub*)  $\rightarrow$  [https://www.gnu.org/software/jitter/manual/structured.html](#)
    - JitterLisp manual (*a little better*)  $\rightarrow$  [https://www.gnu.org/software/jitter/manual/jitterlisp.html](#)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (**existing as an empty stub**) — however the sources are very easy to understand
    - JitterLisp manual (**a little better**)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (**existing as an empty stub**) — however the sources are very easy to understand
    - JitterLisp manual (**a little better**)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (**existing as an empty stub**) — however **the sources are very easy to understand**
    - JitterLisp manual (**a little better**)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial>
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (**existing as an empty stub**) — however **the sources are very easy to understand**
    - JitterLisp manual (**a little better**)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial> (**incomplete**)
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`





# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (**existing as an empty stub**) — however **the sources are very easy to understand**
    - JitterLisp manual (**a little better**)
- Tutorial at <http://ageinghacker.net/projects/jitter-tutorial> (**incomplete**)
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as `lucasaiu`: Libera network, `##jitter`, `##epsilon`, and `#poke`



# How to learn about GNU Jitter, as of March 2022

- Web page at <https://gnu.org/software/jitter>
- Jitter Texinfo documentation ( $\Rightarrow$  Info, PDF, HTML, ...):
  - The Jitter manual (**very, very incomplete**)
  - Jitter example manuals:
    - Structured manual (**existing as an empty stub**) — however **the sources are very easy to understand**
    - JitterLisp manual (**a little better**)
- Tutorial at <http://ageinhacker.net/projects/jitter-tutorial> (**incomplete**)
- Contact me:
  - Mailing list: see the web page
  - I am always on IRC as [lucasaiu](#): Libera network, [##jitter](#), [##epsilon](#), and [#poke](#)



# The end

<https://www.gnu.org/software/jitter>  
<http://ageinghacker.net>

# Thanks.

Any questions?



# The end



<https://www.gnu.org/software/jitter>  
<http://ageinghacker.net>

# Thanks.

Any questions?




# Bibliography I

-  Ertl, M. A. and Gregg, D. (2004). Retargeting JIT compilers by using C-compiler generated executable code. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, PACT '04*, pages 41–50, Washington, DC, USA. IEEE Computer Society.
-  Saiu, L. (2017). The art of the language VM or Machine-generating virtual machine code or Almost zero overhead with almost zero assembly or My virtual machine is faster than yours. GNU Hackers' Meeting 2017, Knüllwald-Niederbeisheim, Germany, August 2017. The first public presentation about Jitter, still useful as an introduction. Slides and video recording available from <http://ageinhacker.net/talks>.



# Bibliography II

-  Saiu, L. (2021). Informal jitter talk. Informal live presentation, March 2021. A friendly talk including a live demo, mostly improvised and not particularly well prepared, with friends from the GNU poke project. Video recording available from <https://archive.org/details/jitter-presentation--2021-03-25>.

