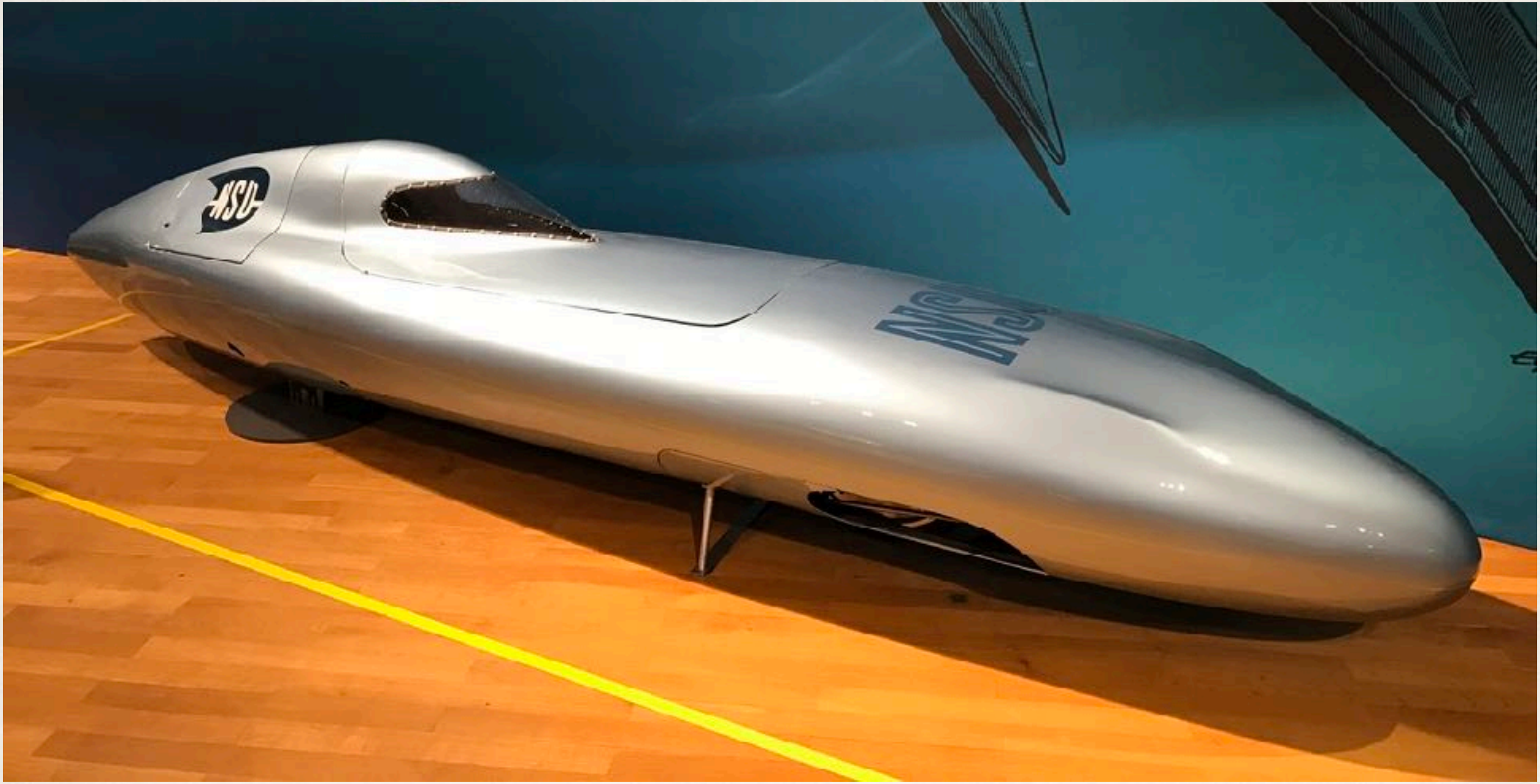


Design versus Performance:
From Giotto via the Embedded Machine to Selfie

Christoph Kirsch, University of Salzburg, Austria

LCTES 2017, Barcelona, Spain

Joint Work

- Giotto/E-Machine:
  Arkadeb Ghosal, Thomas Henzinger, Ben Horowitz, Daniel Iercan, Rupak Majumdar, Marco Sanvido

- Selfie:
  Alireza Abyaneh, Martin Aigner, Sebastian Arming, Christian Barthel, Michael Lippautz, Cornelia Mayer, Simone Oblasser

Inspiration

- Armin Biere: SAT Solvers
- Donald Knuth: Art
- Jochen Liedtke: Microkernels
- David Patterson: RISC
- Niklaus Wirth: Compilers

I am always interested in the <u>slowest</u> design

And maybe even the most memory- and energy-consuming

I call this the logical baseline

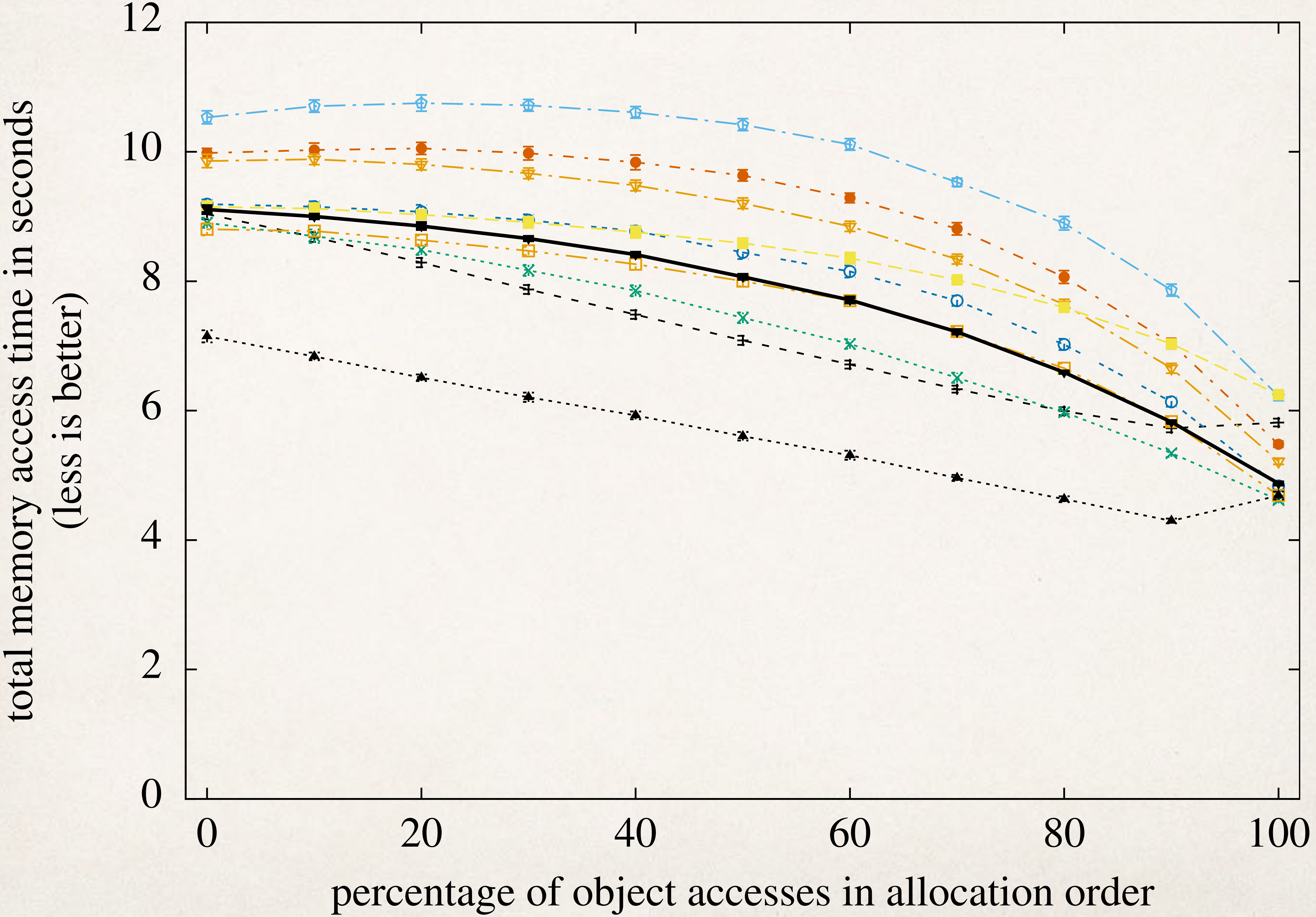
It helps to understand the problem

How many lines of code do you need to implement a SAT solver?

I am also interested in the fastest, “optimal” baseline

Logical baselines are real
and may not be that easy to find

Optimal baselines are often hypothetical
and may not be that easy to find as well


12
10
8
6
4
2
0
0
20
40
60
80
100
total memory access time in seconds
(less is better)
percentage of object accesses in allocation order

We engineer between
logical and optimal baselines
but often forget what’s good enough

Complexity may be unavoidable
but usually there is a lot of choice
where to put it

Three Examples

- Giotto @ EMSOFT 2001 (Proc. of the IEEE 2003)
    - Real-Time Scheduling
    - Synchronous Reactive Languages
- The Embedded Machine @ PLDI 2002 (TOPLAS 2007)
    - Interpreters, Emulators, Virtual Machines
- Selfie @ Onward! 2017 (conditionally accepted)

Giotto: The Problem in Early 2000

determinism

predictability

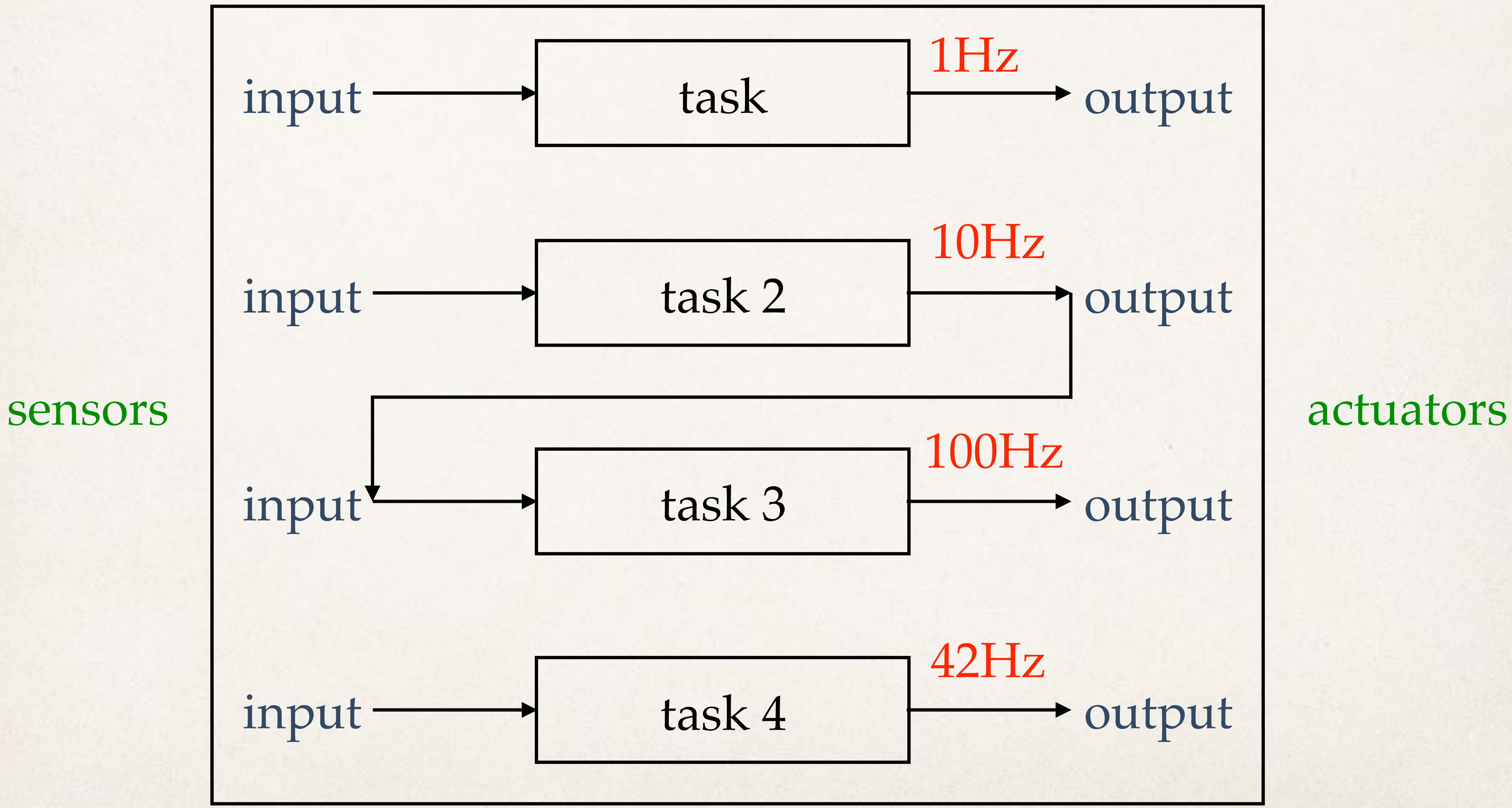
Programming real-time control software on distributed systems of embedded computers

efficiency

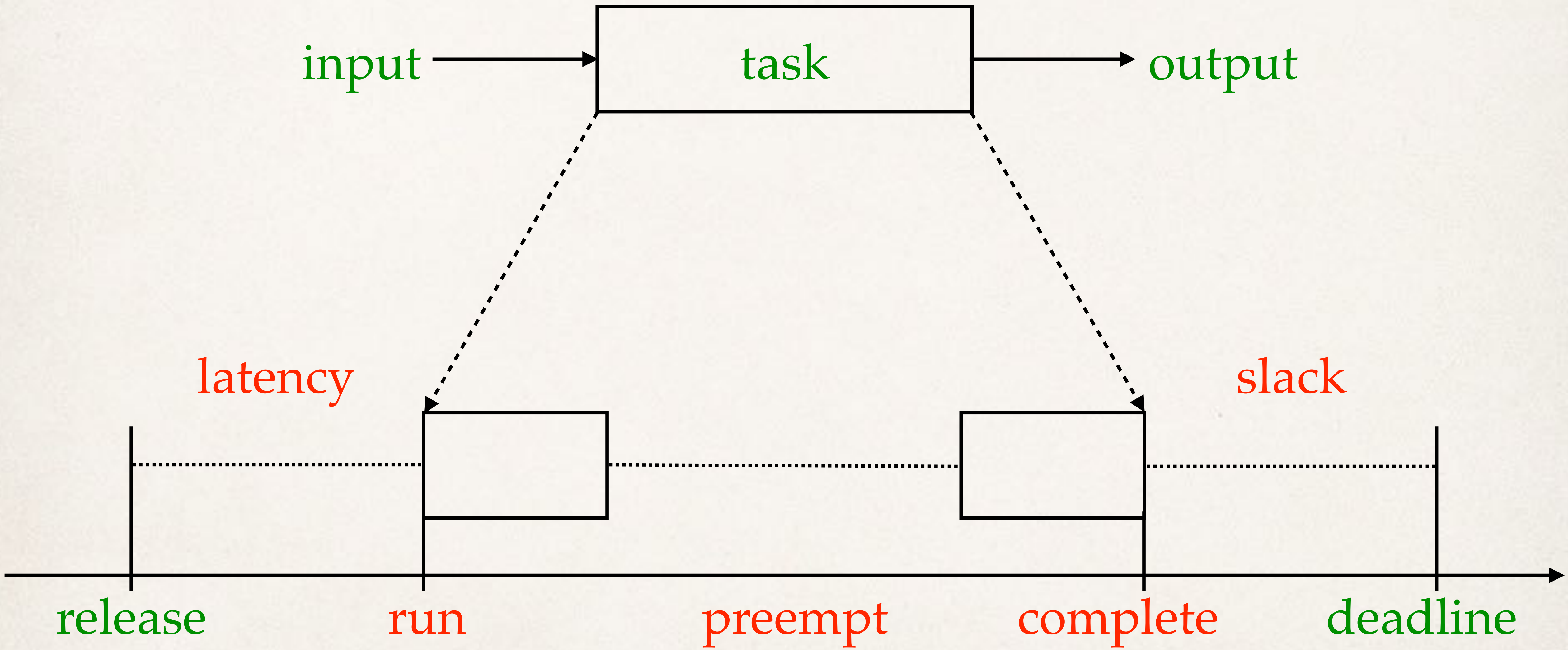
portability

maintainability

Real-Time Task Model



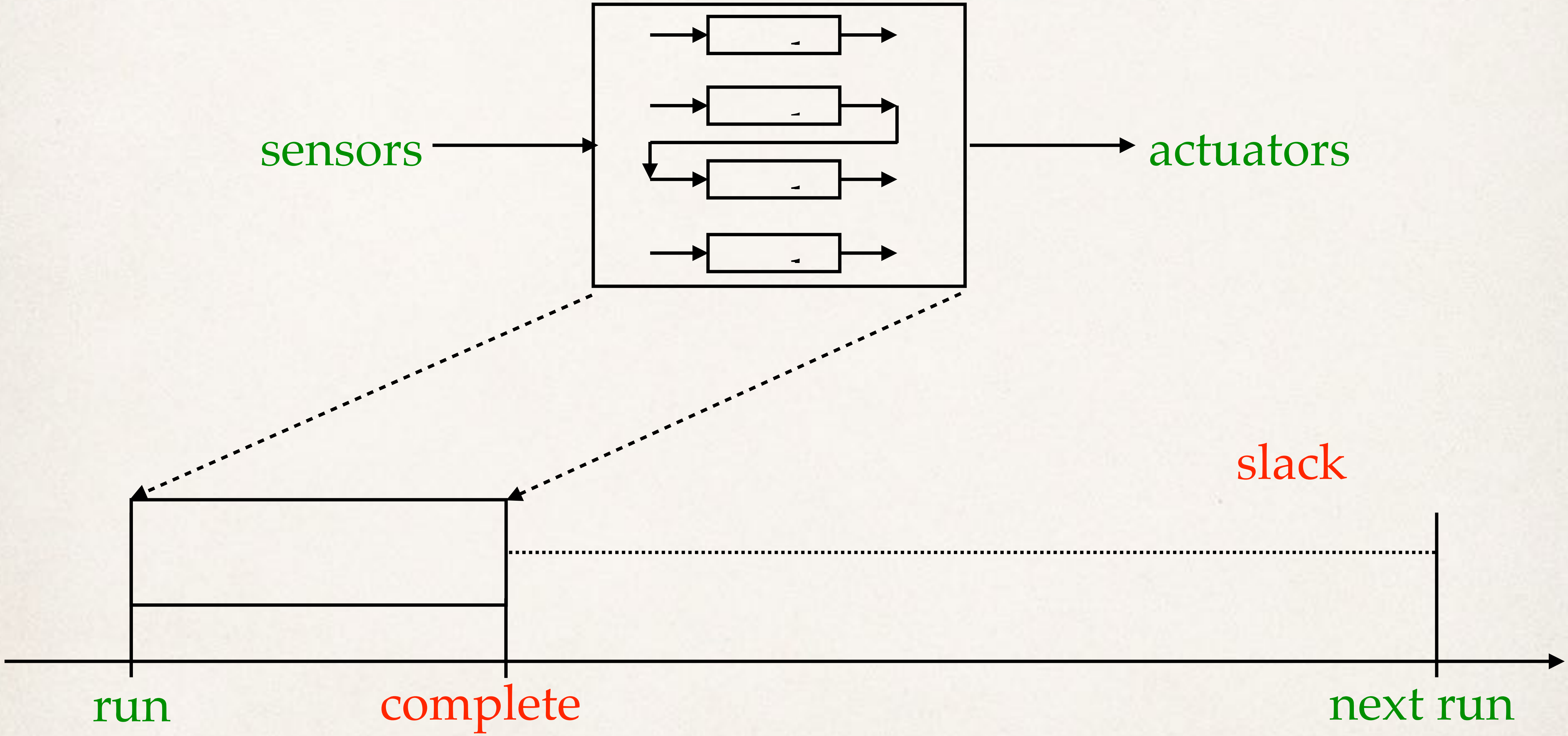
Real-Time Scheduling



Bounded Execution Time (BET) Model w/ R. Sengupta, 2007

Things need to be done before the deadline

Synchronous Reactive Programming



Zero Execution Time (ZET) Model w/ R. Sengupta, 2007

Things need to be done before the next event communication delays

failures

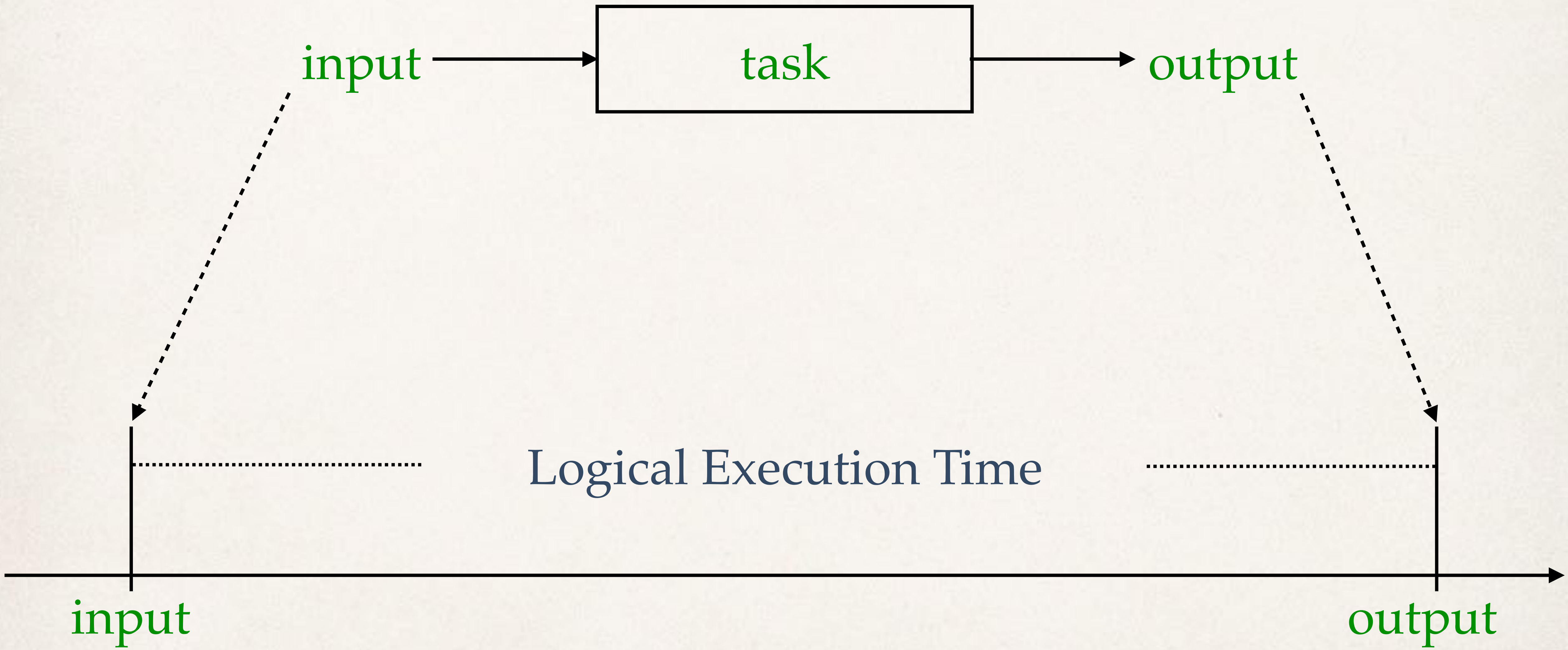
Distributed Embedded Computers

heterogeneous

10-100s

multicore

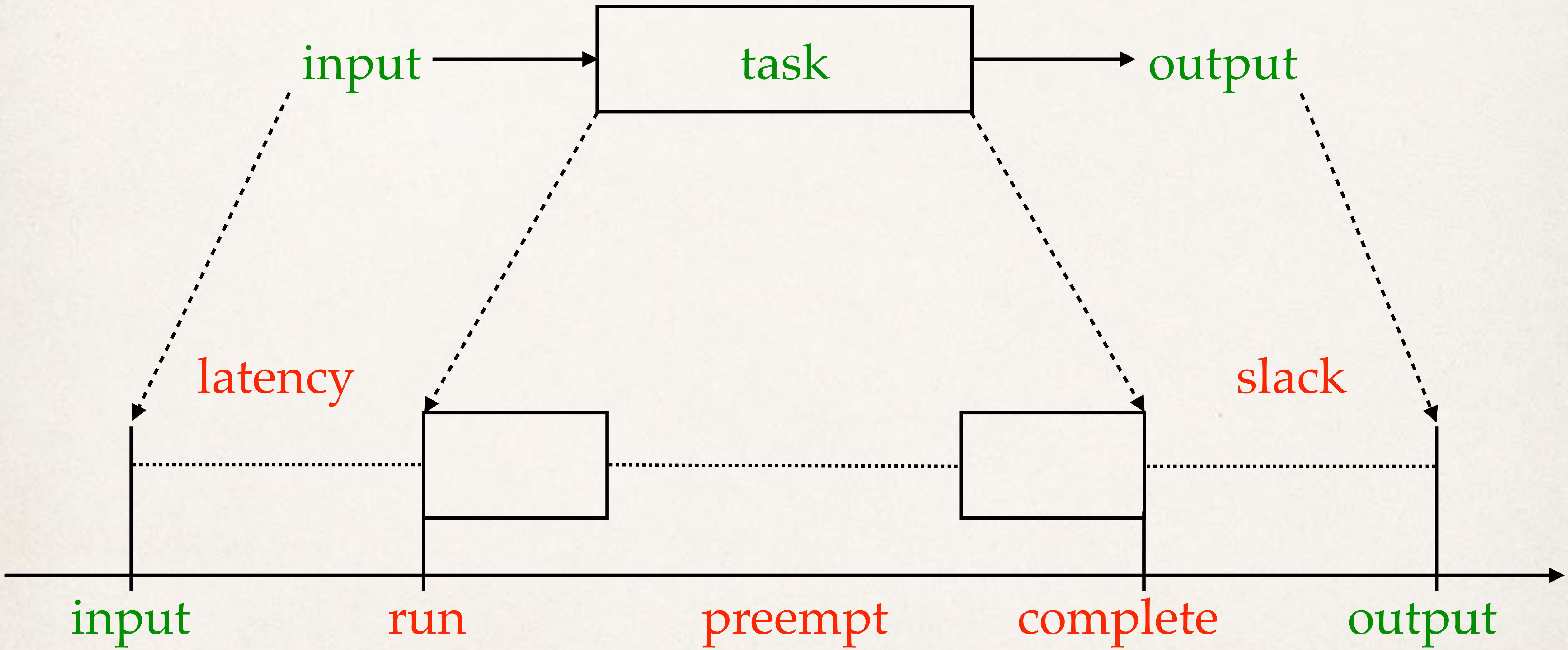
Logical Execution Time (LET) w/ T.A. Henzinger, B. Horowitz @ EMSOFT 2001



cf. Physical Execution Time (PET) Model w/ R. Sengupta, 2007

Program as if there is enough (CPU) time,
just like if there is enough memory

Giotto @ EMSOFT 2001



The compiler and runtime system check
if there is enough time

We call that checking time safety

incremental compilation

separate compilation

Time-safe Giotto programs are <u>time-deterministic</u>

[EMSOFT 2001, Proc. of the IEEE 2003]

compositional scheduling

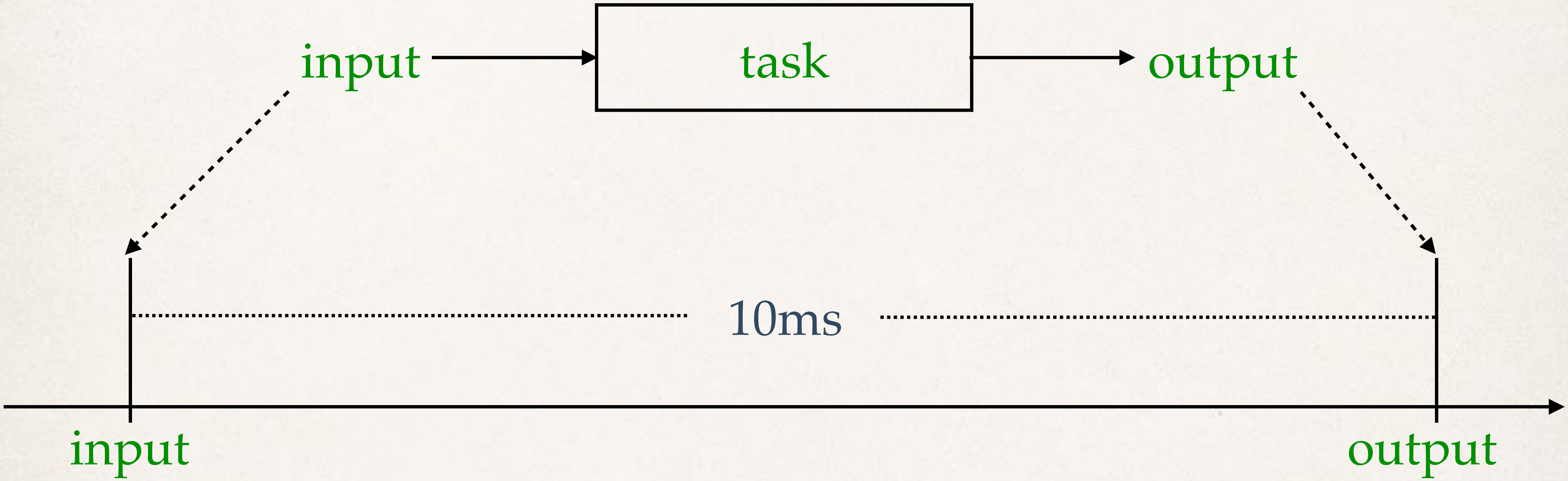
distributed scheduling

Rather than being as fast as possible
we try to be as predictable as possible
and use (CPU) time to do other things

How do we compile
a domain-specific language like Giotto?

Let’s take a detour via PLDI and TOPLAS and work on a target machine first

The Embedded Machine w/ T.A. Henzinger @ PLDI 2002/TOPLAS 2007



```
A: write output
   read input
   release task
   jump to A@10ms
```

```
A: write output
   read input
   release task
   jump to A@10ms
```

dynamic linking

dynamic loading

Time-safe E code is <u>time-deterministic</u>

[PLDI 2002, TOPLAS 2007]

exception handling

schedule-carrying code

Rather than being as <u>fast</u> as possible
we try to be as <u>portable</u> as possible and
again use (CPU) time to do <u>other</u> things

Design versus Performance?

Giotto @ EMSOFT

2001

E Machine @ PLDI

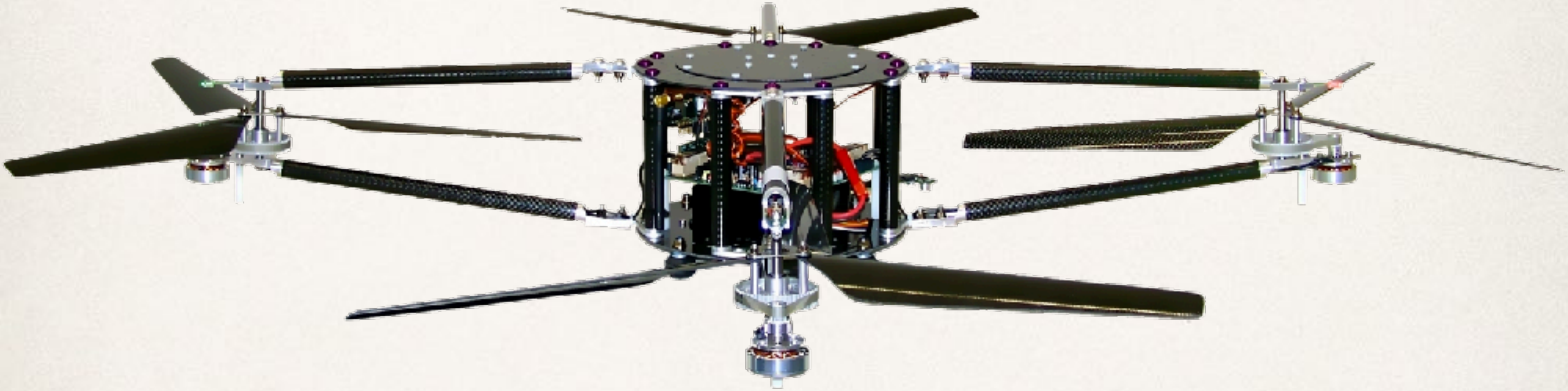
2002

…

Selfie @ Onward?

2017

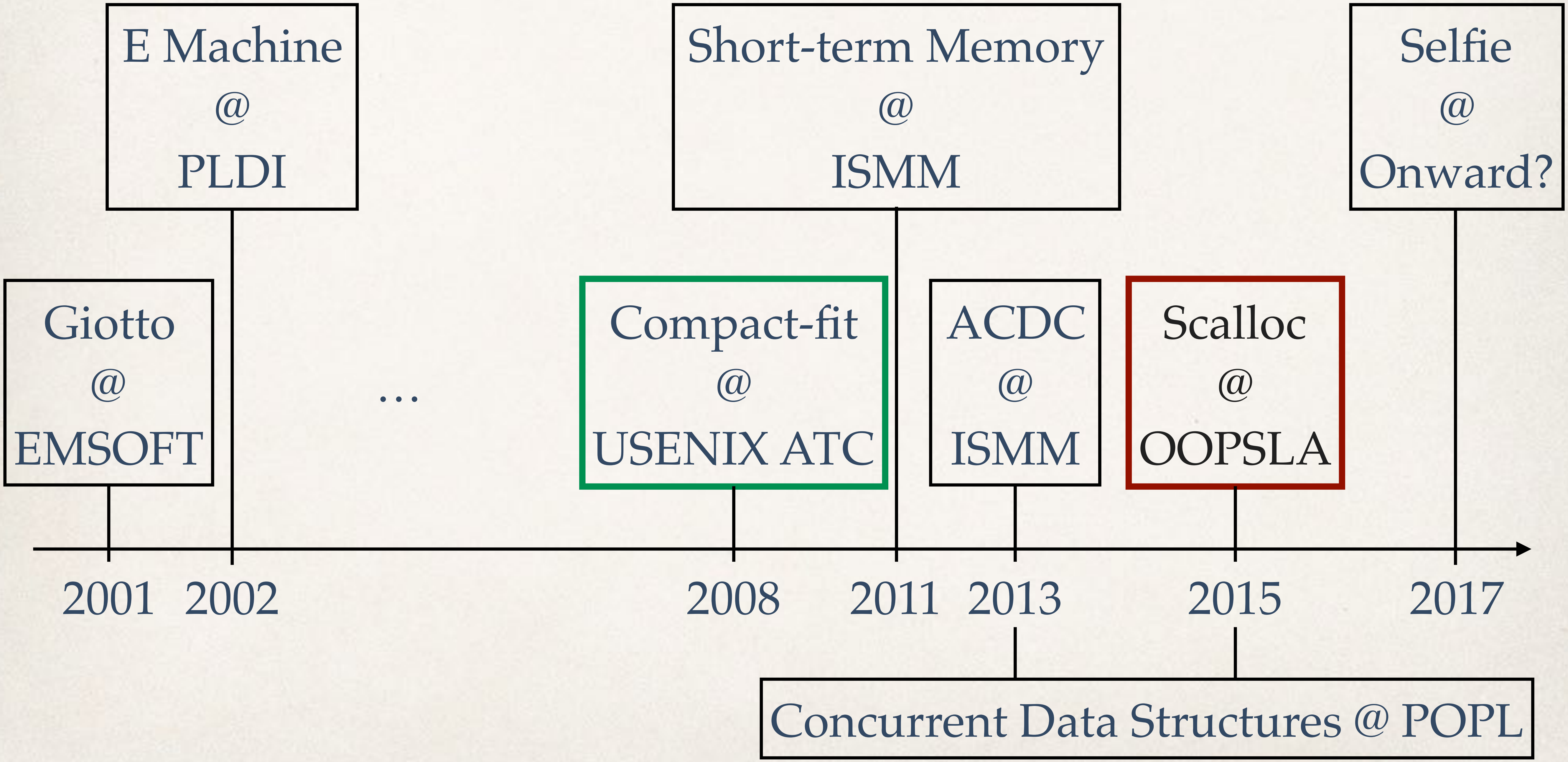
The JAviator @ AIAA GNC 2008



javiator.cs.uni-salzburg.at

w/ R. Trummer et al. @ U. Salzburg and D.F. Bacon et al. @ IBM Hawthorne

Memory Management!



E Machine @ PLDI

Short-term Memory @ ISMM

Selfie @ Onward?

Giotto @ EMSOFT

…

Compact-fit @ USENIX ATC

ACDC @ ISMM

Scalloc @ OOPSLA

2001 2002 2008 2011 2013 2015 2017

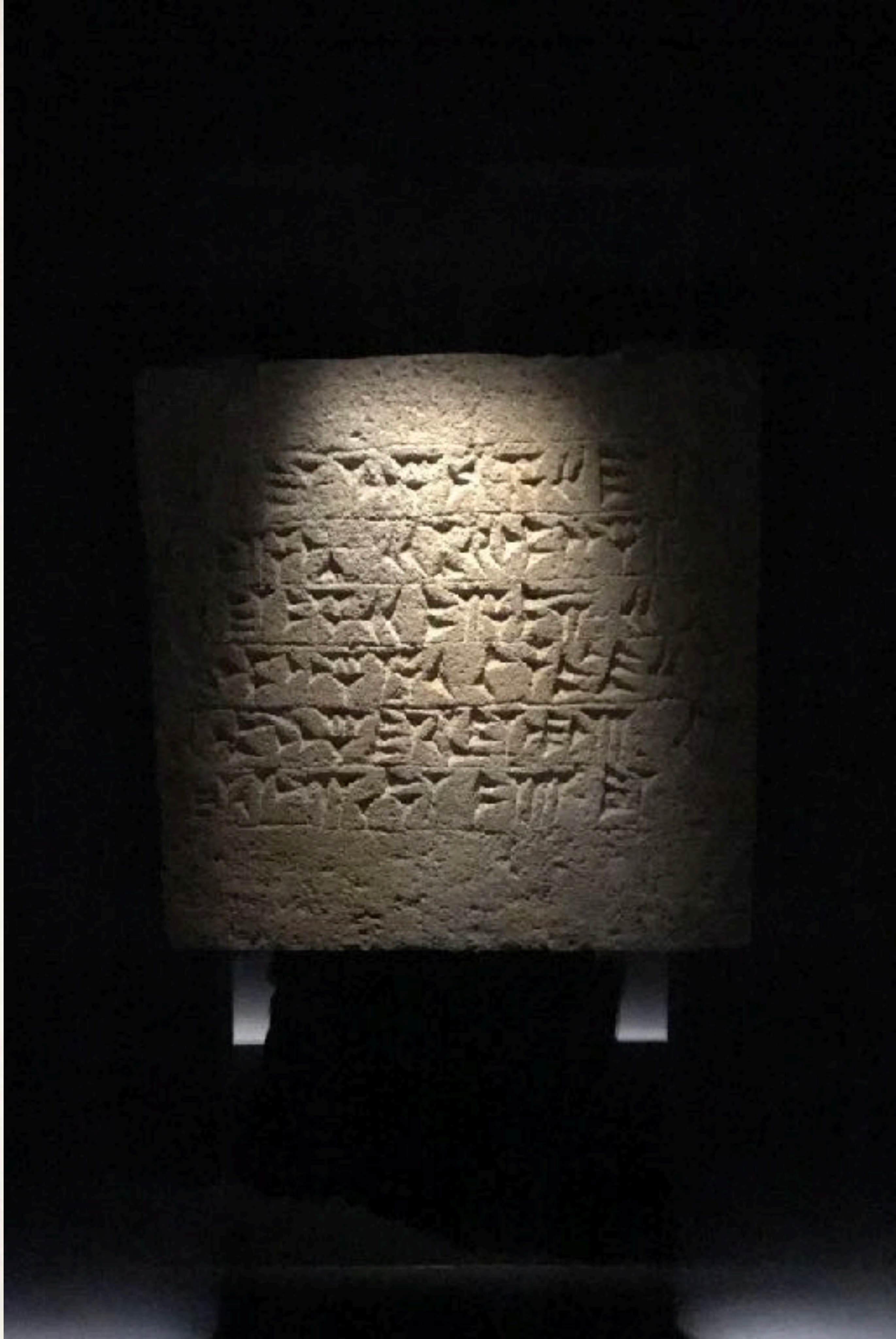
Concurrent Data Structures @ POPL

What else can we slow down?

Teaching Computer Science from First Principles!

What is the meaning of this sentence?

Selfie as in self-referentiality



Interpretation

Translation

Teaching the Construction of Semantics of Formalisms

Virtualization

Verification

Selfie: Teaching Computer Science [selfie.cs.uni-salzburg.at]

- *Selfie* is a self-referential 7k-line C implementation (in a single file) of:

  1. a self-compiling compiler called *starc* that compiles a tiny subset of C called C Star (C*) to a tiny subset of MIPS32 called MIPSter,
  2. a self-executing emulator called *mipster* that executes MIPSter code including itself when compiled with starc,
  3. a self-hosting hypervisor called *hypster* that virtualizes mipster and can host all of selfie including itself,
  4. a tiny C* library called *libcstar* utilized by all of selfie, and
  5. a tiny, experimental SAT solver called *babysat*.

Website

selfie.cs.uni-salzburg.at

Book (Draft)

leanpub.com/selfie

Code

github.com/cksystemsteaching/selfie

Discussion of Selfie recently reached 3rd place on Hacker News

news.ycombinator.com

nsf.gov/csforall

code.org

computingatschool.org.uk

programbydesign.org

bootstrapworld.org

k12cs.org

csfieldguide.org.nz

5 statements:
assignment
while
if
return
procedure()

```
int atoi(int *s) {
    int i;
    int n;
    int c;

    i = 0;
    n = 0;
    c = *(s+i);

    while (c != 0) {
        n = n * 10 + c - '0';
        if (n < 0)
            return -1;

        i = i + 1;
        c = *(s+i);
    }

    return n;
}
```

no data structures, just `int` and `int*` and dereferencing: the * operator

character literals
string literals

integer arithmetics
pointer arithmetics

no bitwise operators
no Boolean operators

library: `exit`, `malloc`, `open`, `read`, `write`

Scarcity versus Abundance

If you want structs implement them!

Selfie and the Basics

Library	1. Building Selfie	1. Semantics
	2. Encoding C* Literals	2. Encoding
	3. Program/Machine State	3. State
Compiler	4. C*/Command Line Scanners	4. Regularity
	5. C* Parser and Procedures	5. Stack
	6. Symbol Table and the Heap	6. Name
	7. MIPSter Code Generator	7. Time
Emulator	8. Memory Management	8. Memory
Hypervisor	9. Composite Data Types	9. Type
SAT Solver	10. MIPSter Boot Loader	10. Bootstrapping
	11. MIPSter Emulator	11. Interpretation
`selfie.c`	12. MIPSter Hypervisor	12. Virtualization

Rather than being as fast as possible we try to be as simple as possible and hopefully find new synergies

```
> make
cc -w -m32 -D'main(a,b)=main(a,char**argv)' selfie.c -o selfie
```

bootstrapping `selfie.c` *into x86* `selfie` *executable*
using standard C compiler

(now also available for RISC-V machines)

```
> ./selfie
./selfie: usage: selfie { -c { source } | -o binary | -s assembly
| -l binary } [ ( -m | -d | -y | -min | -mob ) size ... ]
```

selfie usage

```
> ./selfie -c selfie.c

./selfie: this is selfie's starc compiling selfie.c

./selfie: 176408 characters read in 7083 lines and 969 comments
./selfie: with 97779(55.55%) characters in 28914 actual symbols
./selfie: 261 global variables, 289 procedures, 450 string literals
./selfie: 1958 calls, 723 assignments, 57 while, 572 if, 243 return
./selfie: 121660 bytes generated with 28779 instructions and 6544
bytes of data
```

compiling `selfie.c` *with x86* `selfie` *executable*

(takes seconds)

```
> ./selfie -c selfie.c -m 2 -c selfie.c

./selfie: this is selfie's starc compiling selfie.c

./selfie: this is selfie's mipster executing selfie.c with 2MB of
physical memory

selfie.c: this is selfie's starc compiling selfie.c

selfie.c: exiting with exit code 0 and 1.05MB of mallocated memory

./selfie: this is selfie's mipster terminating selfie.c with exit code
0 and 1.16MB of mapped memory
```

compiling `selfie.c` *with x86* `selfie` *executable into a MIPSter executable*

and

then running that MIPSter executable to compile `selfie.c` *again*

(***takes ~6 minutes***)

```
> ./selfie -c selfie.c -o selfie1.m -m 2 -c selfie.c -o selfie2.m

./selfie: this is selfie's starc compiling selfie.c
./selfie: 121660 bytes with 28779 instructions and 6544 bytes of data
written into selfie1.m

./selfie: this is selfie's mipster executing selfie1.m with 2MB of
physical memory

selfie1.m: this is selfie's starc compiling selfie.c
selfie1.m: 121660 bytes with 28779 instructions and 6544 bytes of data
written into selfie2.m

selfie1.m: exiting with exit code 0 and 1.05MB of mallocated memory

./selfie: this is selfie's mipster terminating selfie1.m with exit
code 0 and 1.16MB of mapped memory
```

compiling `selfie.c` *into a MIPSter executable* `selfie1.m`

and

then running `selfie1.m` *to compile* `selfie.c`

into another MIPSter executable `selfie2.m`

*(**takes ~6 minutes**)*

```
> ./selfie -c selfie.c -m 2 -c selfie.c -m 2 -c selfie.c
```

compiling `selfie.c` *with x86* `selfie` executable

__and__

then running that executable to compile `selfie.c` *again*

__and__

then running that executable to compile `selfie.c` *again*

*(**takes ~24 hours**)*

> “The OS is an interpreter until people wanted speed.”
>
> *–ck*

```
> ./selfie -c selfie.c -m 2 -c selfie.c -y 2 -c selfie.c
```

compiling `selfie.c` *with x86* `selfie` *executable*

and

then running that executable to compile `selfie.c` *again*

and

*then **hosting** that executable in a virtual machine to compile* `selfie.c` *again*

*(**takes ~12 minutes**)*

“How do we introduce self-model-checking and maybe even self-verification into Selfie?”

[*https://github.com/cksystemsteaching/selfie/tree/vipster*](https://github.com/cksystemsteaching/selfie/tree/vipster)

SMT Solver

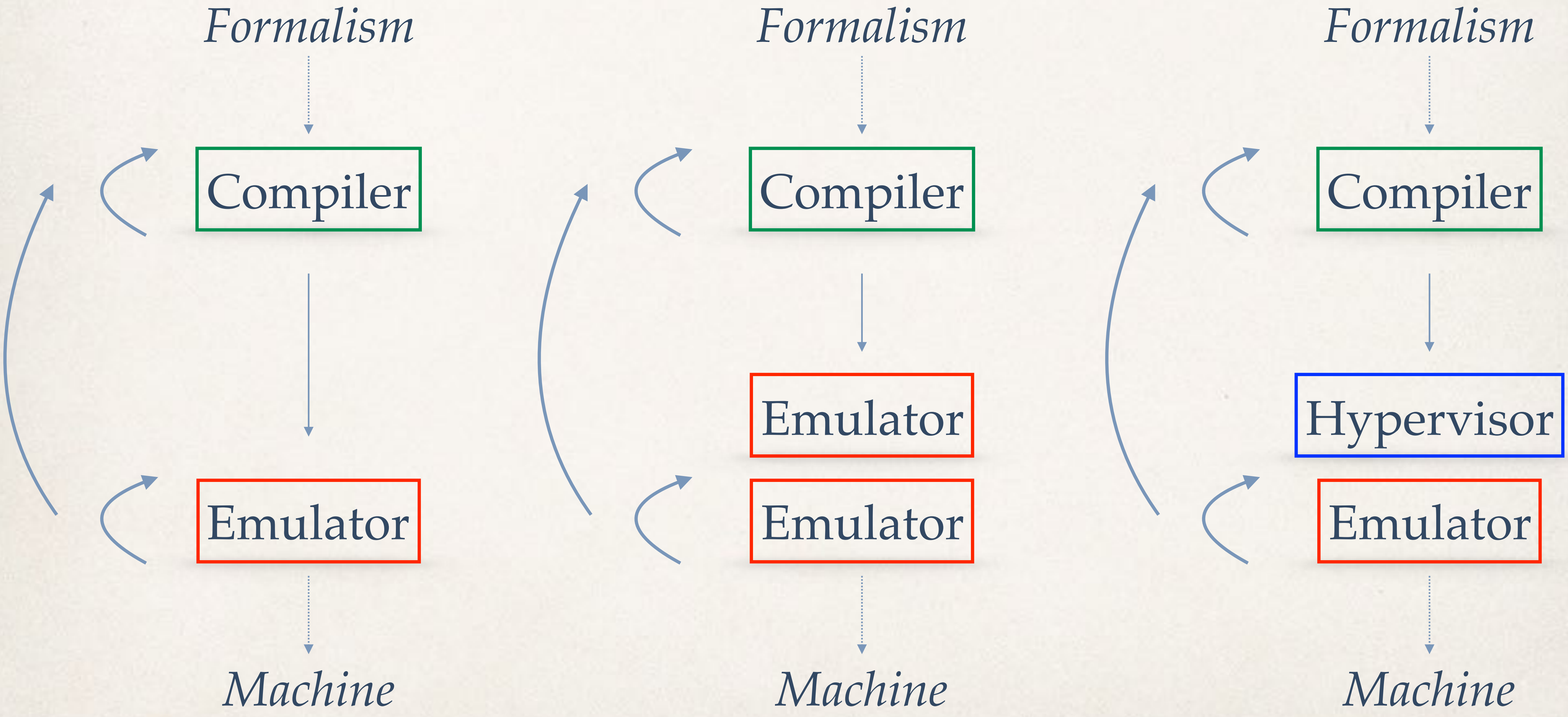
SAT Solver

What is the absolute simplest way of proving non-trivial properties of Selfie using Selfie?

Bounded Model Checker

Inductive Theorem Prover

Semantics and Performance



Formalism

Compiler

Emulator

Machine

Formalism

Compiler

Emulator

Emulator

Machine

Formalism

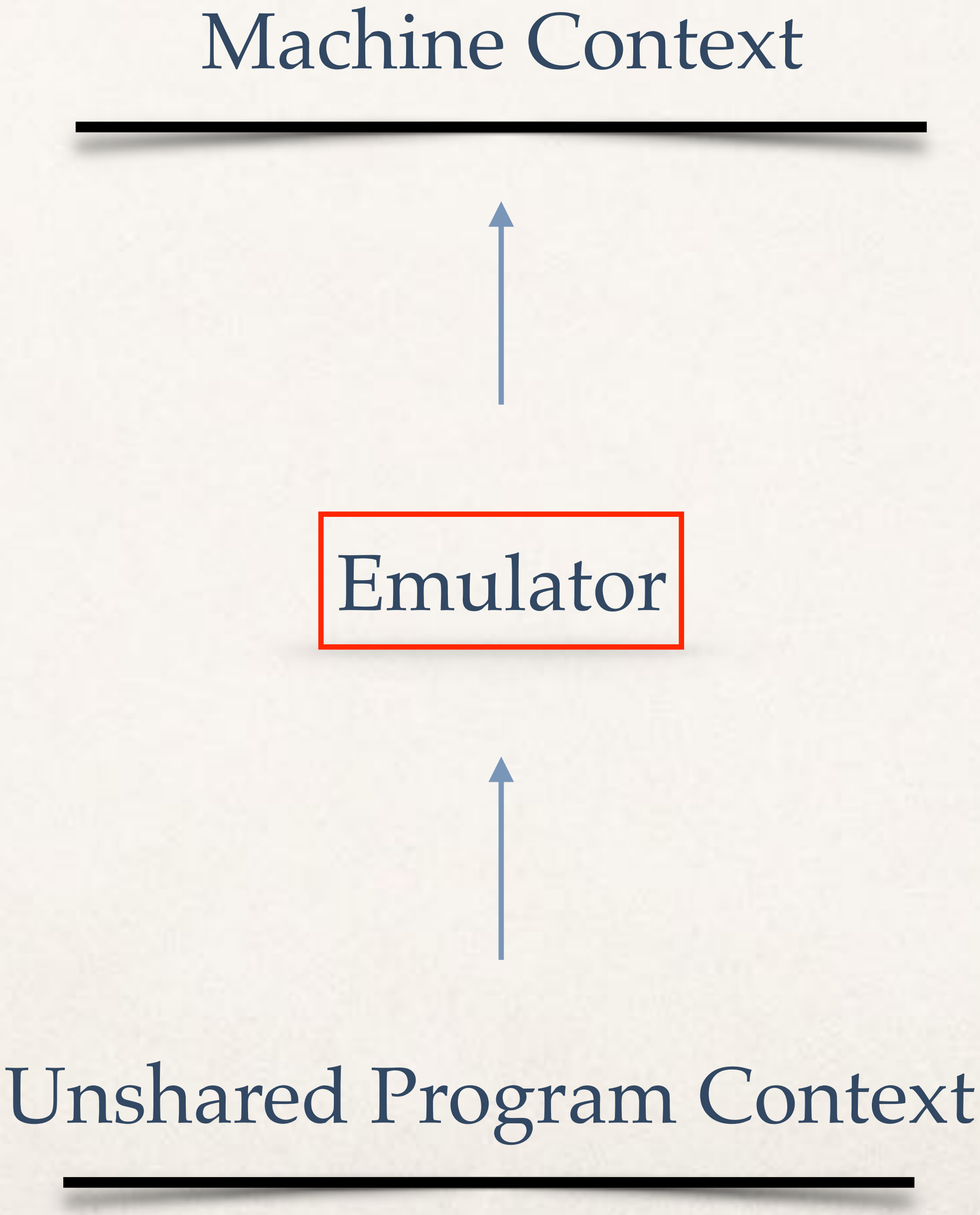
Compiler

Hypervisor

Emulator

Machine

Emulation

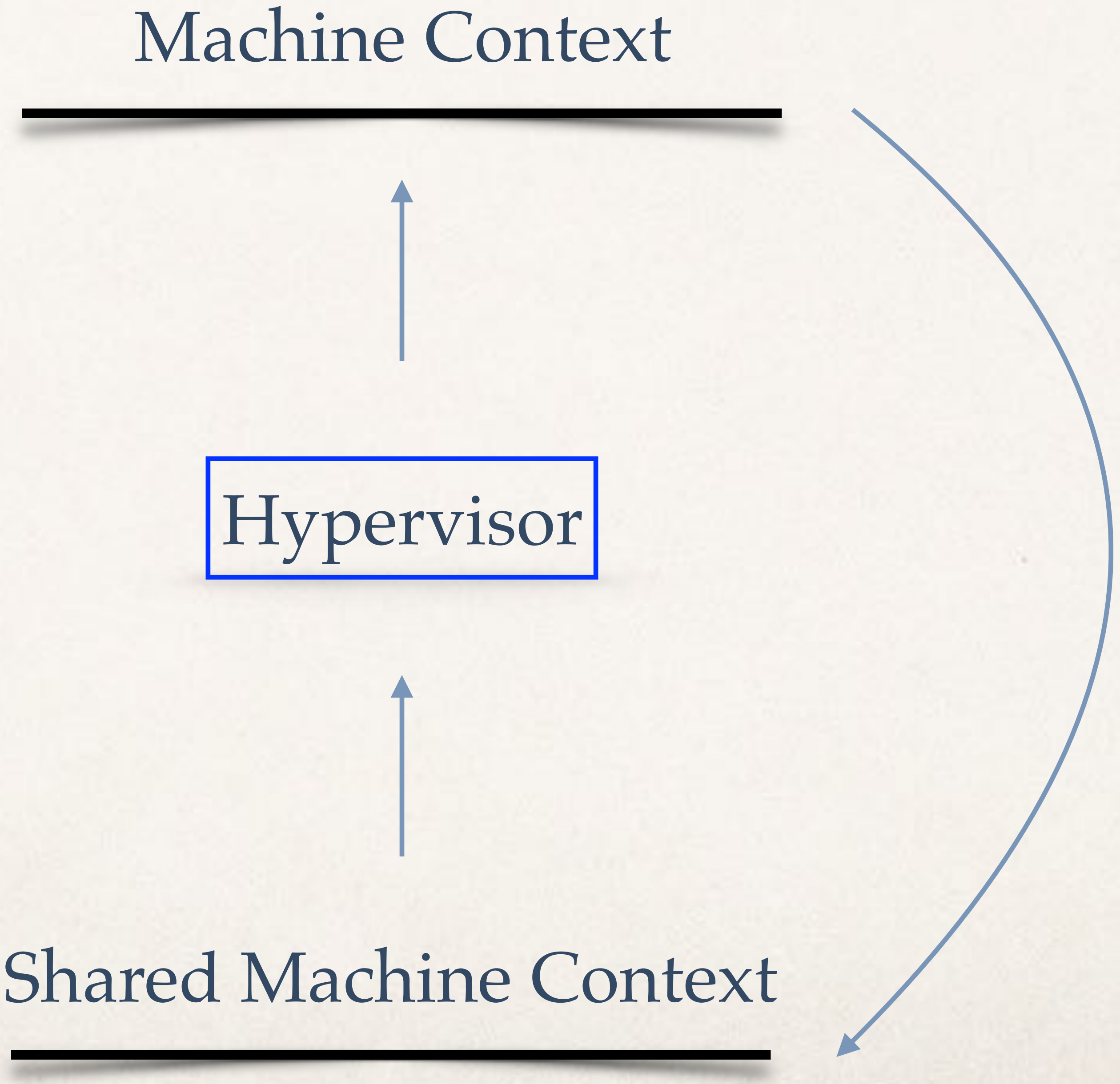


Machine Context

Emulator

Unshared Program Context

Virtualization



Machine Context

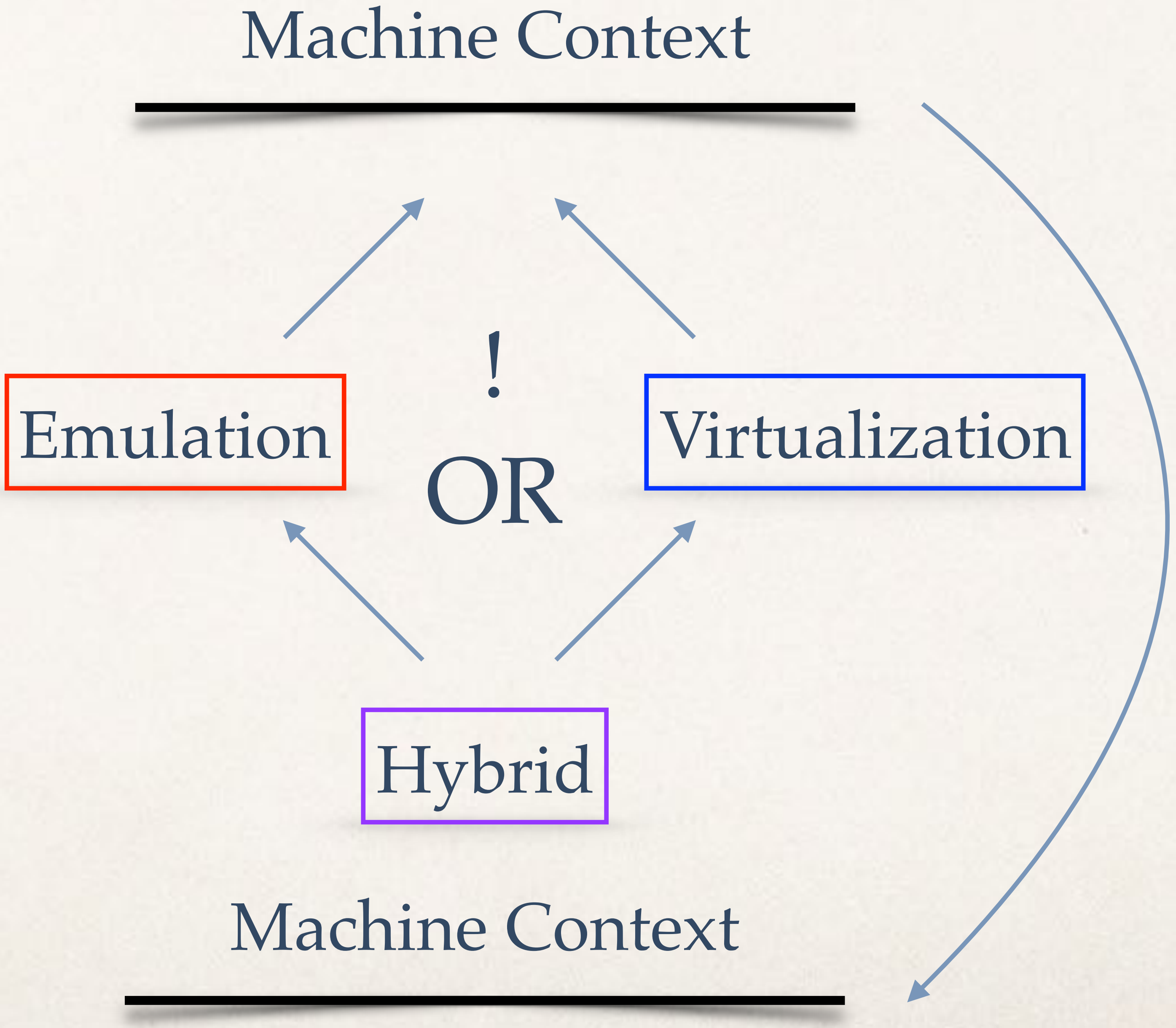
Hypervisor

Shared Machine Context

Proof Obligation

$$\frac{\text{Machine Context}}{\boxed{\text{Emulator}}} \overset{?}{=} \frac{\text{Machine Context}}{\boxed{\text{Hypervisor}}}$$

Hybrid of Emulator & Hypervisor



Machine Context

Emulation

!

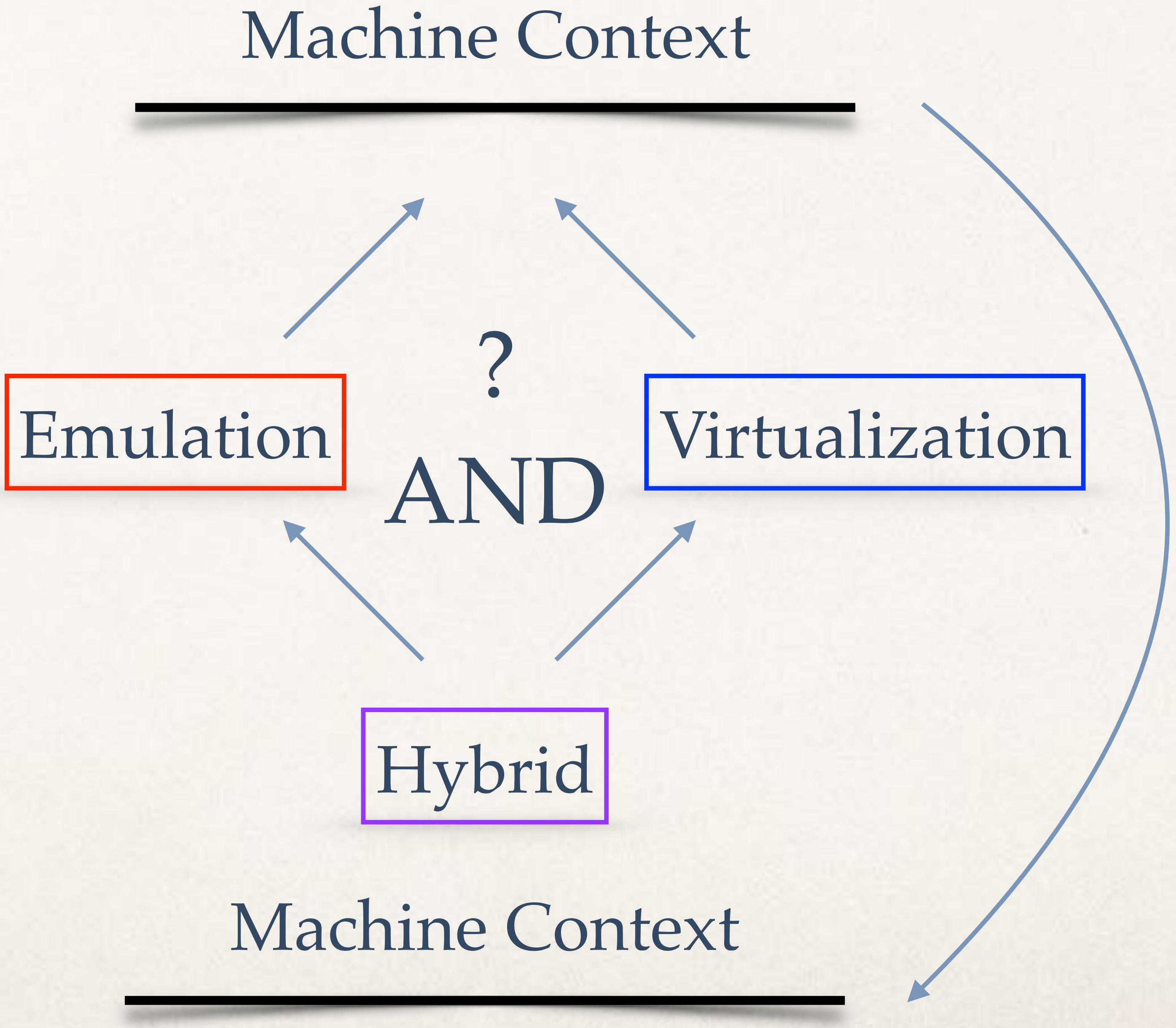
OR

Virtualization

Hybrid

Machine Context

Validation of Functional Equivalence?



Machine Context

?

Emulation

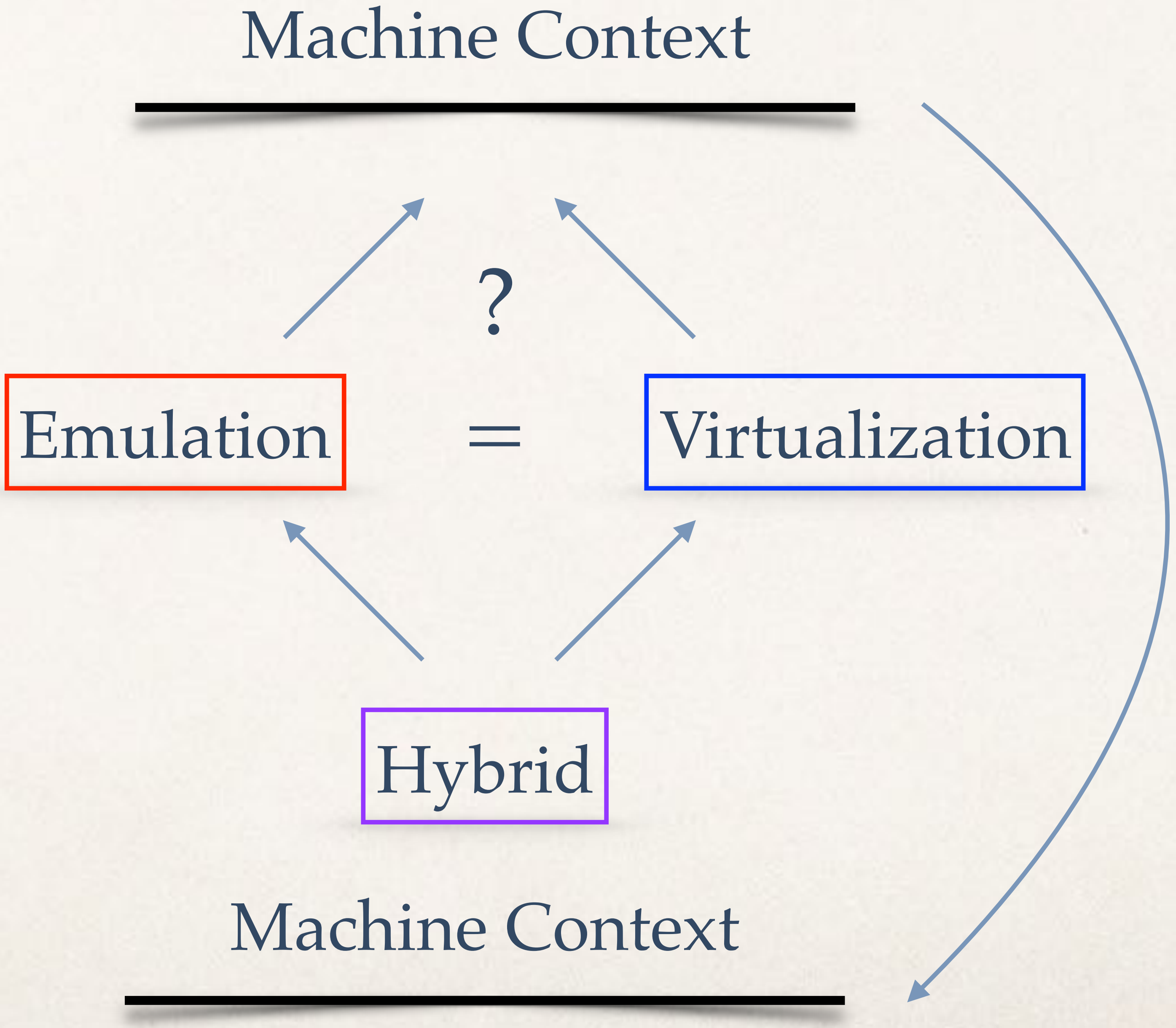
AND

Virtualization

Hybrid

Machine Context

Verification of Functional Equivalence?



Machine Context

?

Emulation

=

Virtualization

Hybrid

Machine Context

Questions

- What are the benefits of the hybrid design in Selfie?
- Will these benefits change the design of real kernels, that is, is the hybrid design realistic?
- Can we develop C* into a useful specification language, cf. ACL2?
- Can we prove interesting properties with a, say, ~10k-line system?
- Will this help teaching rigorous systems and software engineering at bachelor level?
- Will this help identifying basic principles that can be taught to everyone?

Thank you!