KESO

An Open-Source Multi-JVM for Deeply Embedded Systems

Isabella Thomm, **Michael Stilkerich**,
Christian Wawersich, Wolfgang Schröder-Preikschat

Friedrich-Alexander-Universität
Erlangen-Nürnberg

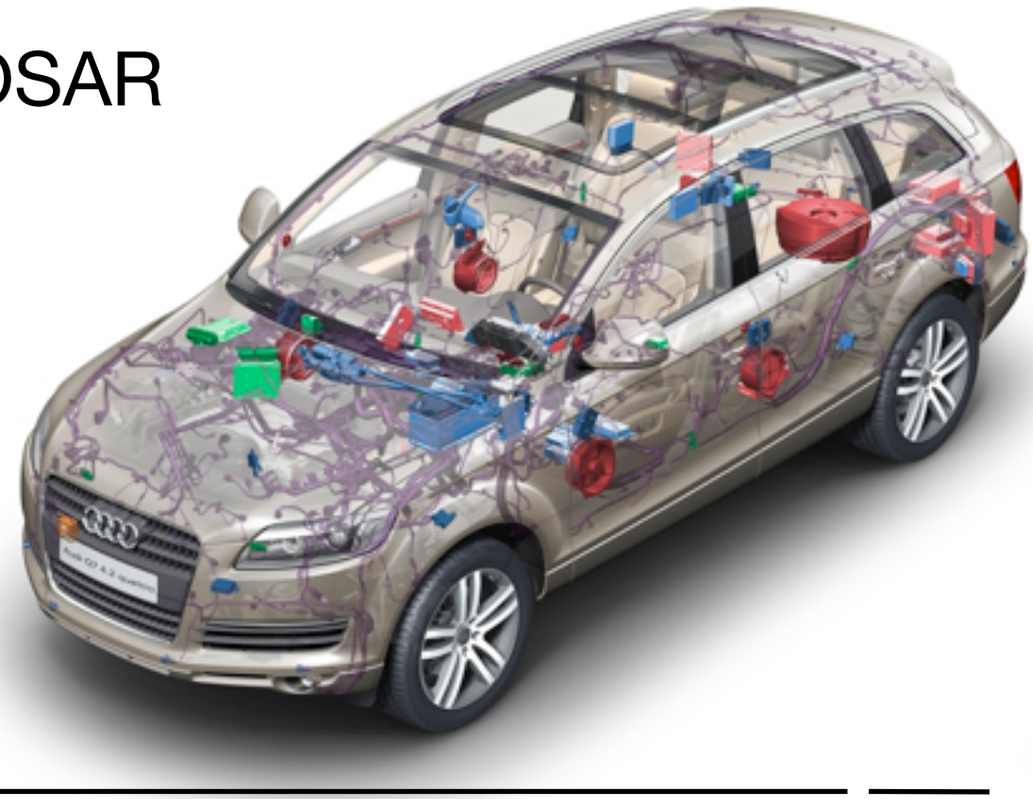
Department of Computer Science 4
Distributed Systems and Operating Systems
Friedrich-Alexander University Erlangen-Nuremberg

http://www4.cs.fau.de/Research/KESO
http://www4.cs.fau.de/~mike
stilkerich@cs.fau.de

Statically-Configured Embedded Systems

- comprehensive a priori knowledge
  - code
  - system objects (tasks/threads, locks, events)
  - system object relationships (e.g., which tasks access which locks)
- example: electronic control units (ECU) in cars
  - mass product ➟ immense cost pressure
  - programming languages: C, C++, Assembler
  - operating system: OSEK/VDX, AUTOSAR
    - static configuration
    - scalability classes
    - **tailored to the application**
  - ECU consolidation
    - application isolation

AUTOSAR



Java and Static Embedded Systems

- benefits
  - more robust software (cf., MISRA-C)
  - software-based spatial isolation
  - infrastructure for new technologies (e.g., state migration)
- problems
  - low-level programming inconvenient/expensive
  - dynamic code loading
    - fully-featured Java runtimes (e.g., J2ME configurations)
  - overhead
    - code is interpreted or JIT compiled (execution time)
    - dynamic linking (footprint)
    - reflection (footprint, analyzability)

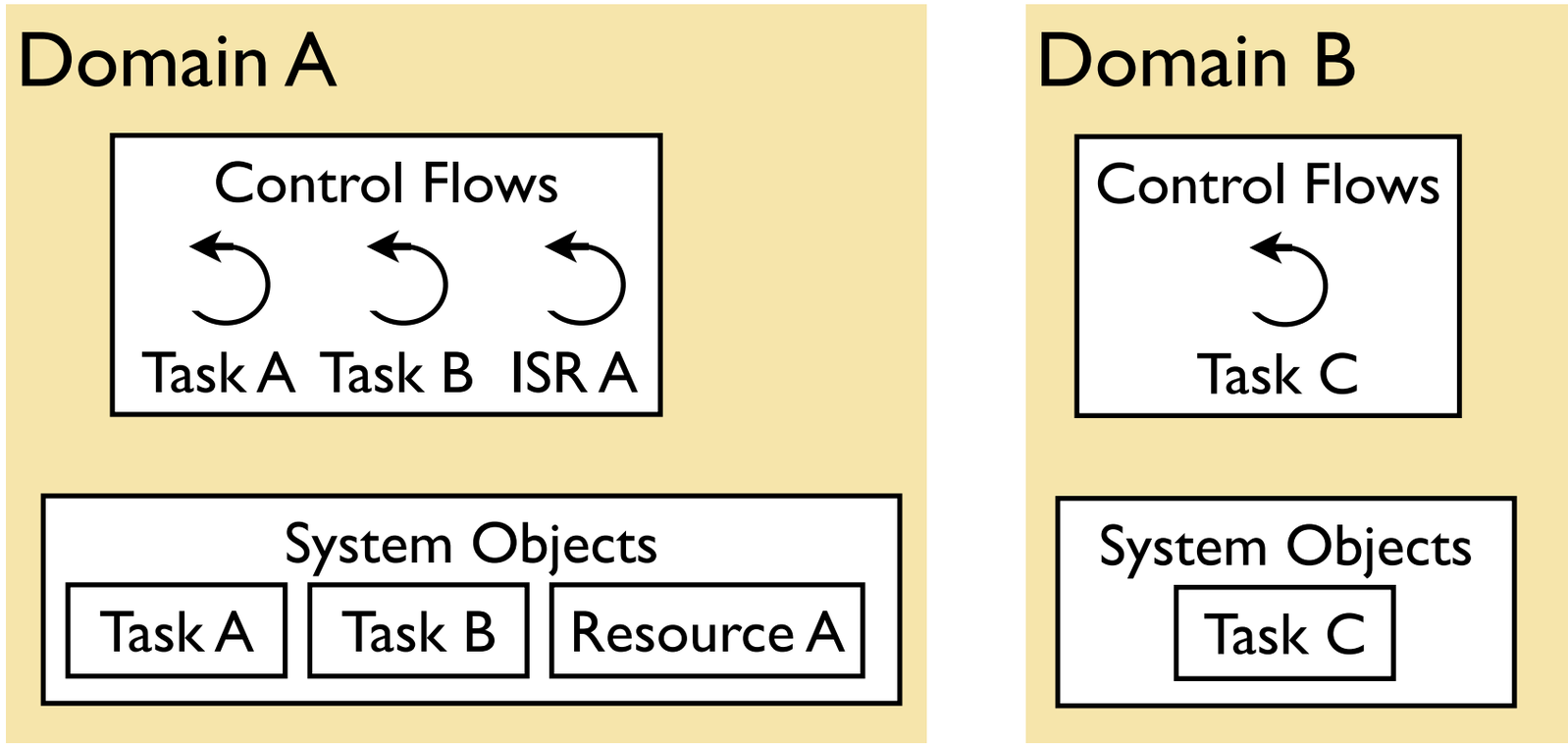
Outline

- Motivation: Statically-Configured Embedded Systems
- KESO: Overview
- Inter-Domain Communication with Portals
- Evaluation

KESO: Overview

- JVM tailoring (instead of fixed configurations)
  - static applications, no dynamic class loading
  - no Java reflection
  - ahead-of-time compilation to Ansi C, VM bundled with application
- scheduling/synchronization provided by underlying OS
  - currently AUTOSAR/OSEK OS
  - accustomed programming model remains
- current limitations
  - simple error hook instead of Java exception handling
  - garbage collector does not bound external fragmentation
  - no support for Java monitor concept

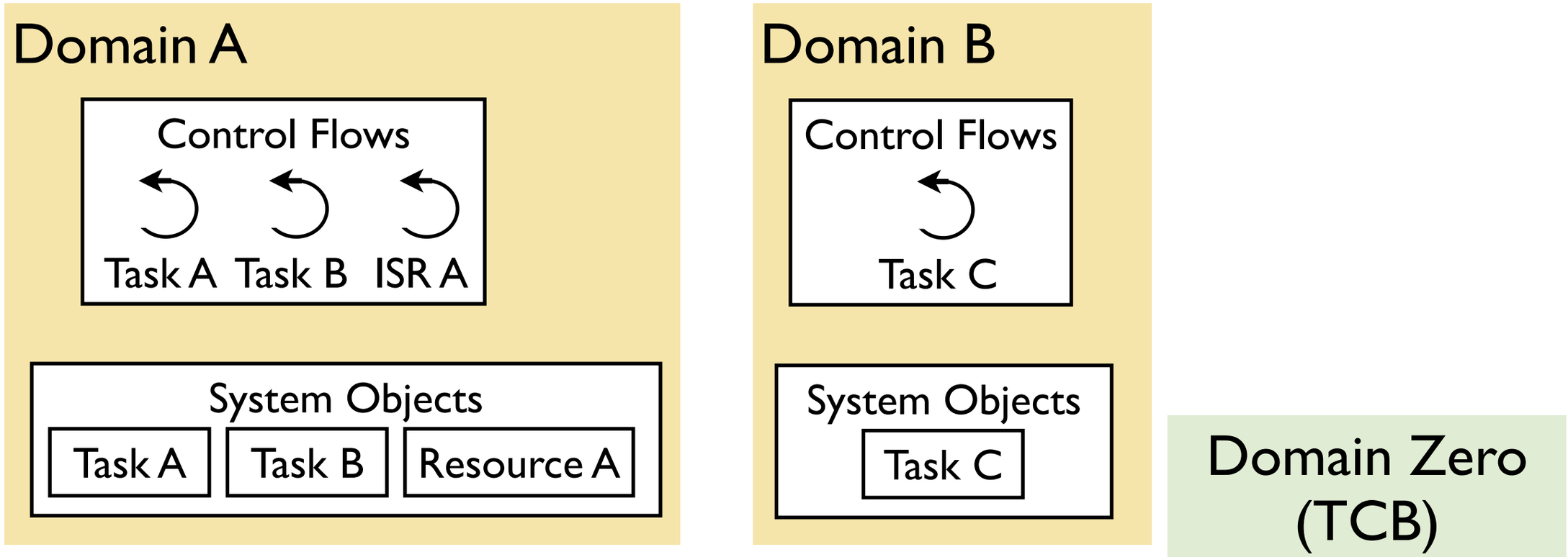
KESO: Architecture



Domains: realms of {memory,service}protection

- containers for control flows and system objects
- appear as a separate JVMs to the application

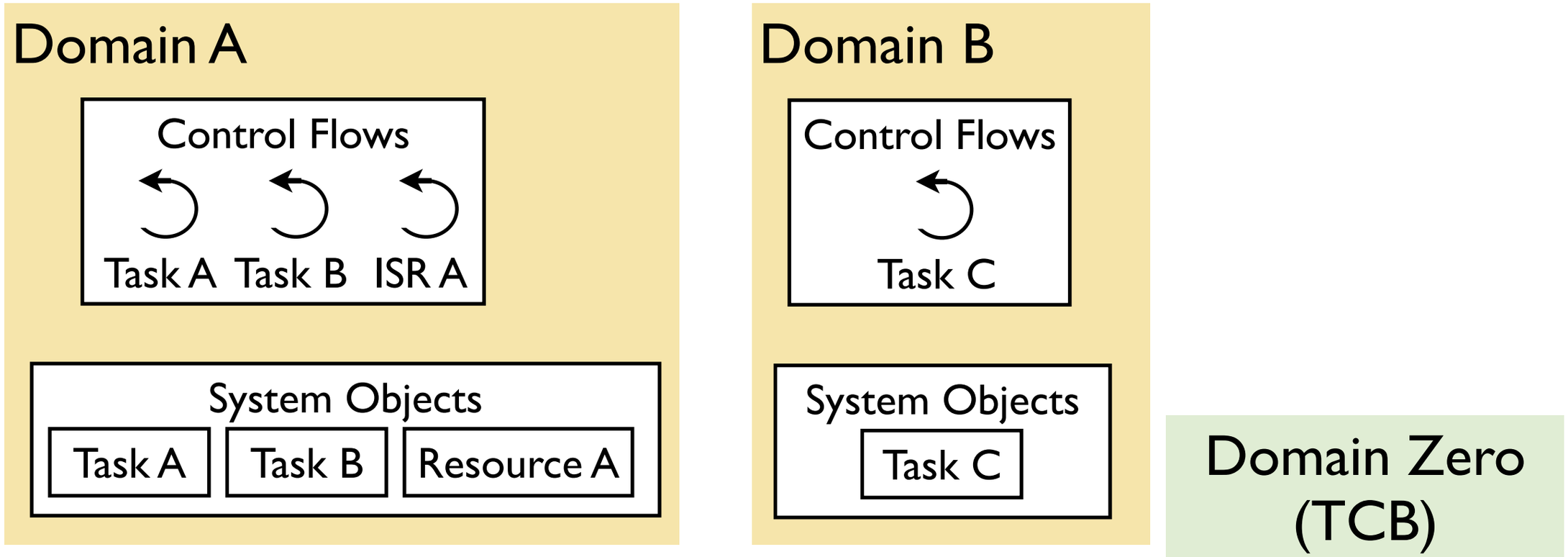
KESO: Architecture



Domain Zero

- trusted control flows of KESO's runtime environment
- currently only the garbage collector

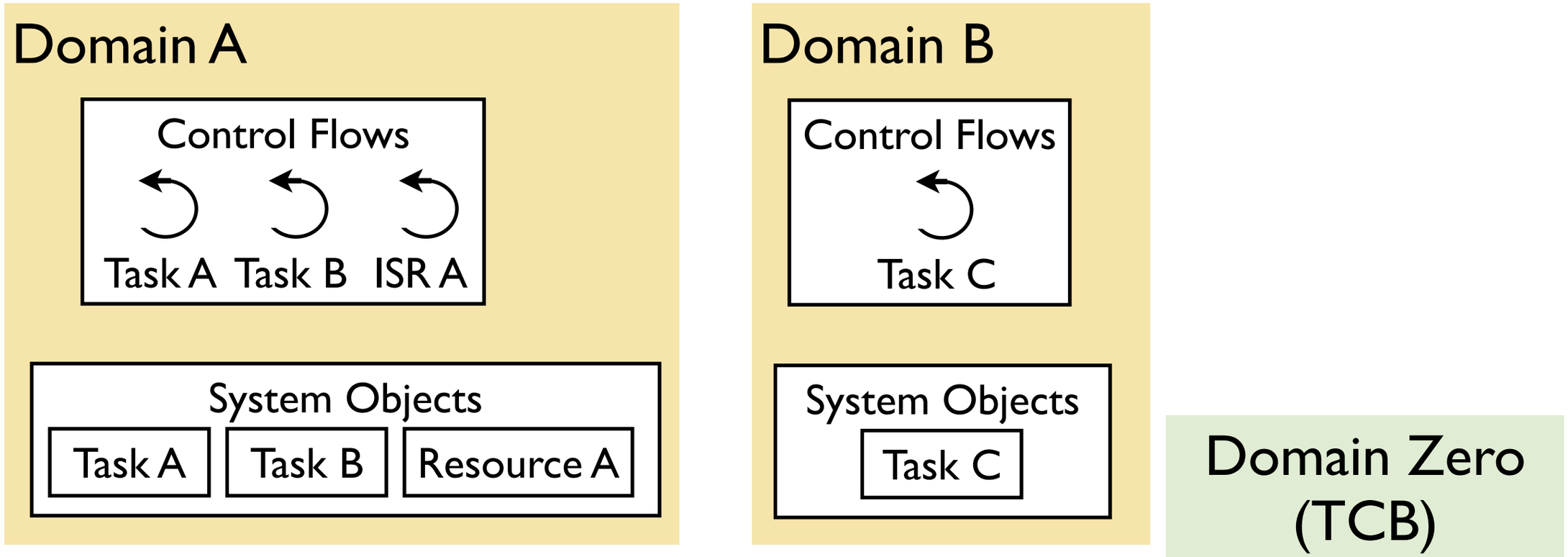
KESO: Architecture



OSEK / AUTOSAR OS

- provides threading/scheduling facilities
- temporal isolation, hardware-based spatial isolation

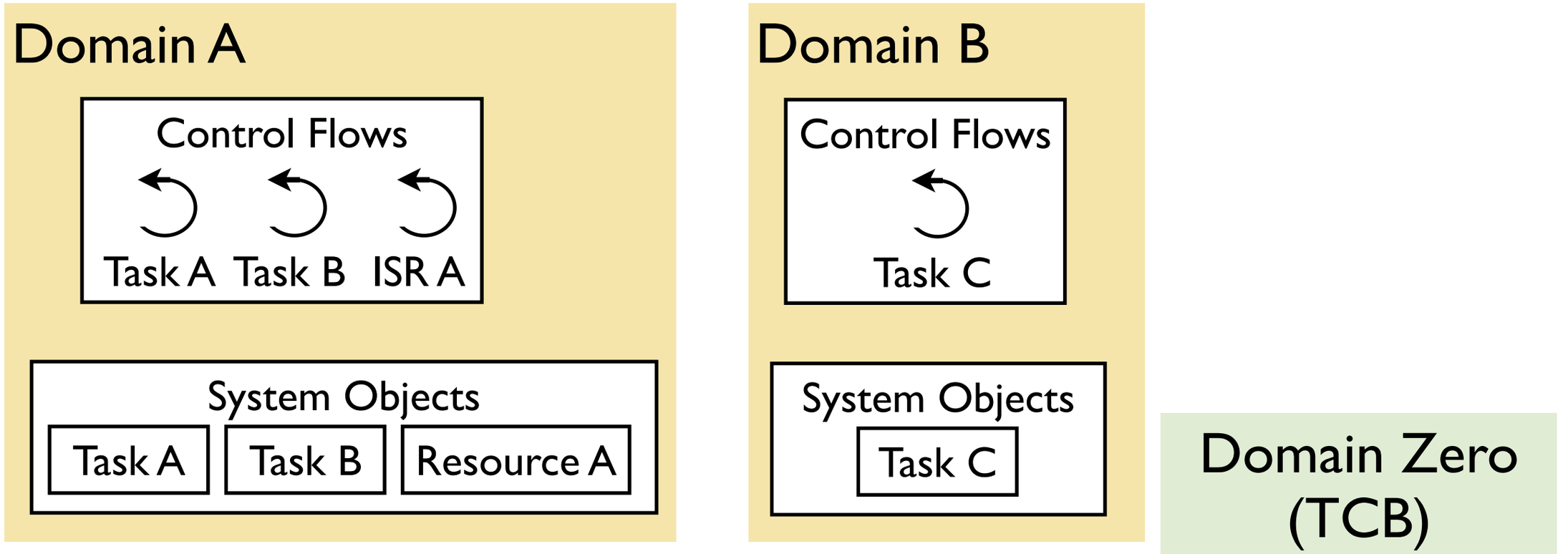
KESO: Architecture



Typical Targets

- low-end: 8-bit AVR (ATmega8535, 8K ROM, 512b RAM)
- higher-end: 32-bit Tricore (TC1796, 2M ROM, 256K RAM)

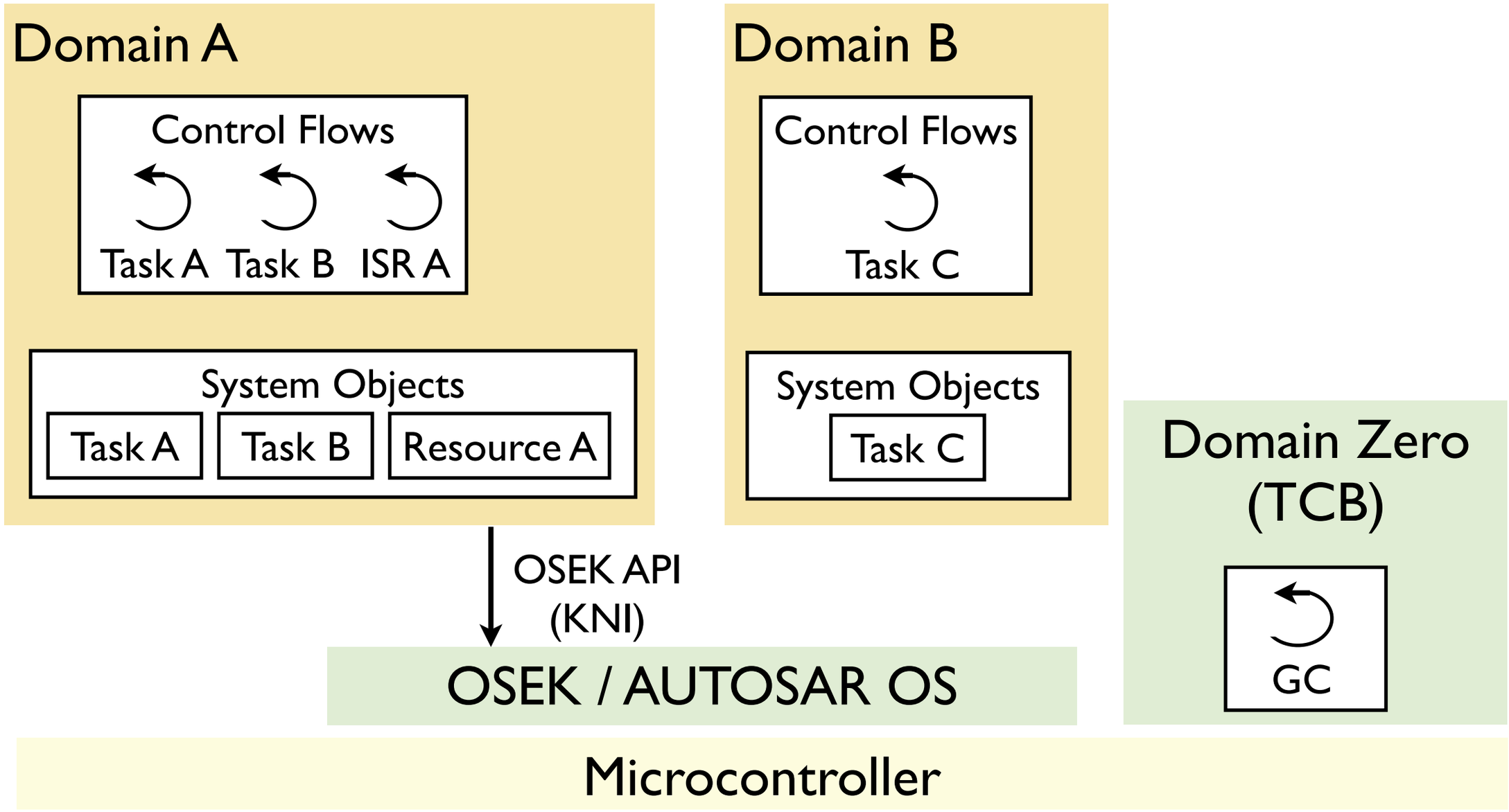
KESO: Architecture



KESO Native Interface (KNI)

- aspect-oriented mechanism for unsafe interactions
- full access to the internal state of Java-to-C compiler

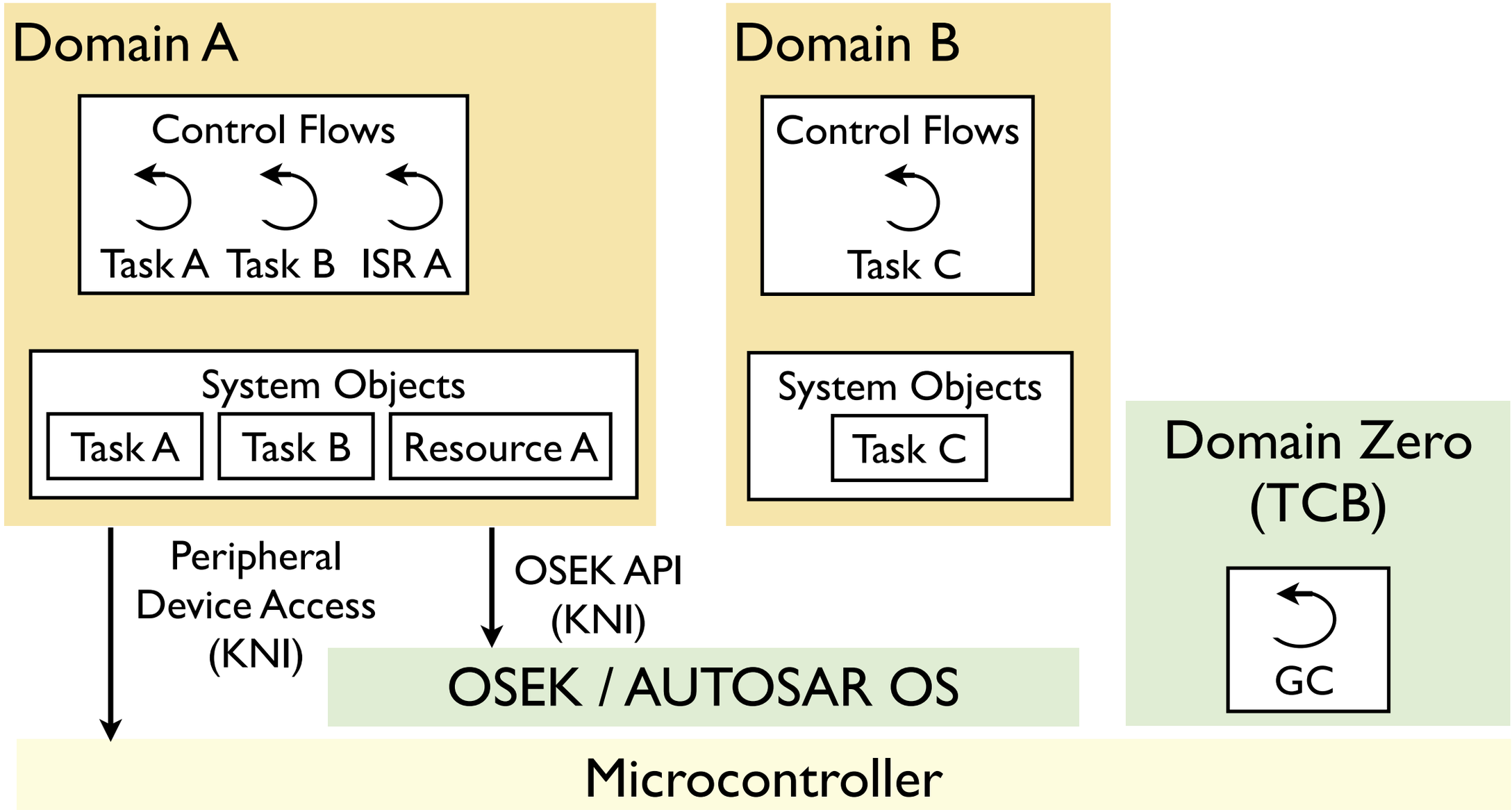
KESO: Architecture



OSEK Java API

- access to OSEK / AUTOSAR system services
- language-based service protection

KESO: Architecture



Peripheral Device Access

- RawMemory (similar to RTSJ)
- Memory-mapped objects (similar to C structs)

Spatial Isolation

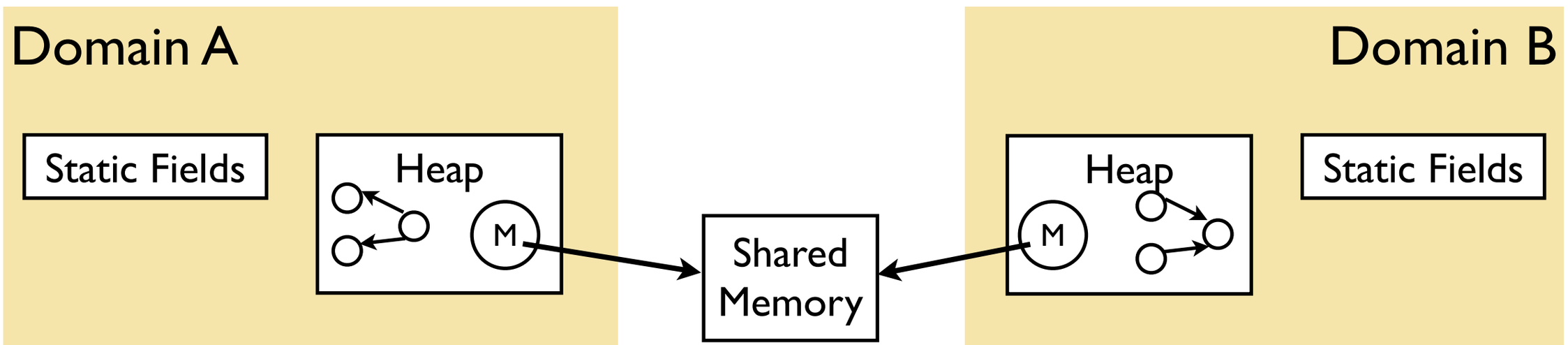
- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated

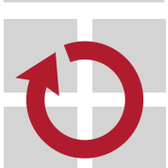
Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated
- Inter-domain Communication

Spatial Isolation

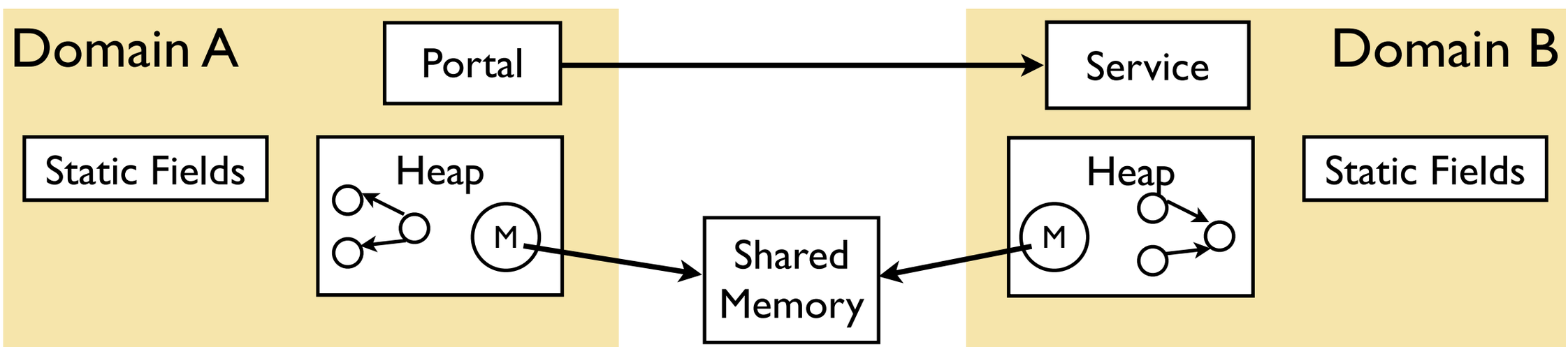
- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated
- Inter-domain Communication
  - Shared Memory (≈ RawMemory with reference counting)





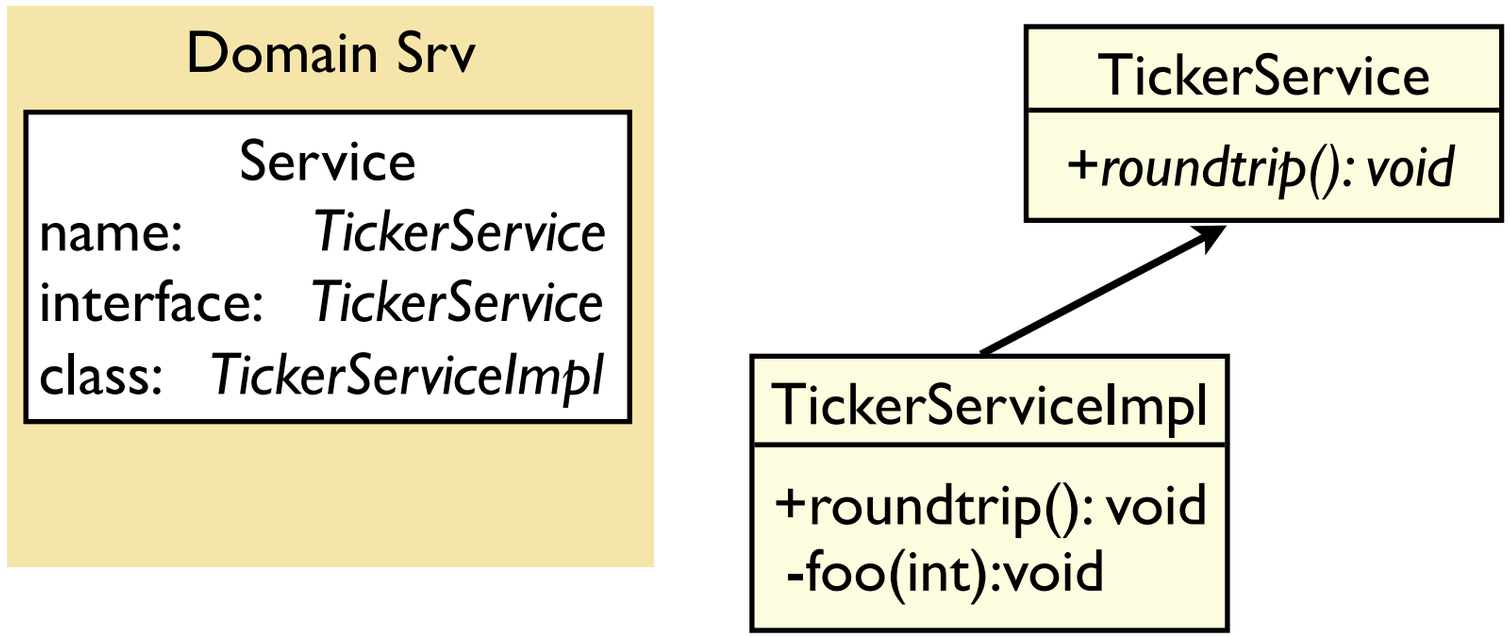
Spatial Isolation

- inhibit shared data among different domains
- own set of static fields in each domain
- logical heap separation (no cross-domain references)
  - current implementation: heaps physically separated
- Inter-domain Communication
  - Shared Memory (≈ RawMemory with reference counting)
  - Portals (RMI-like mechanism)



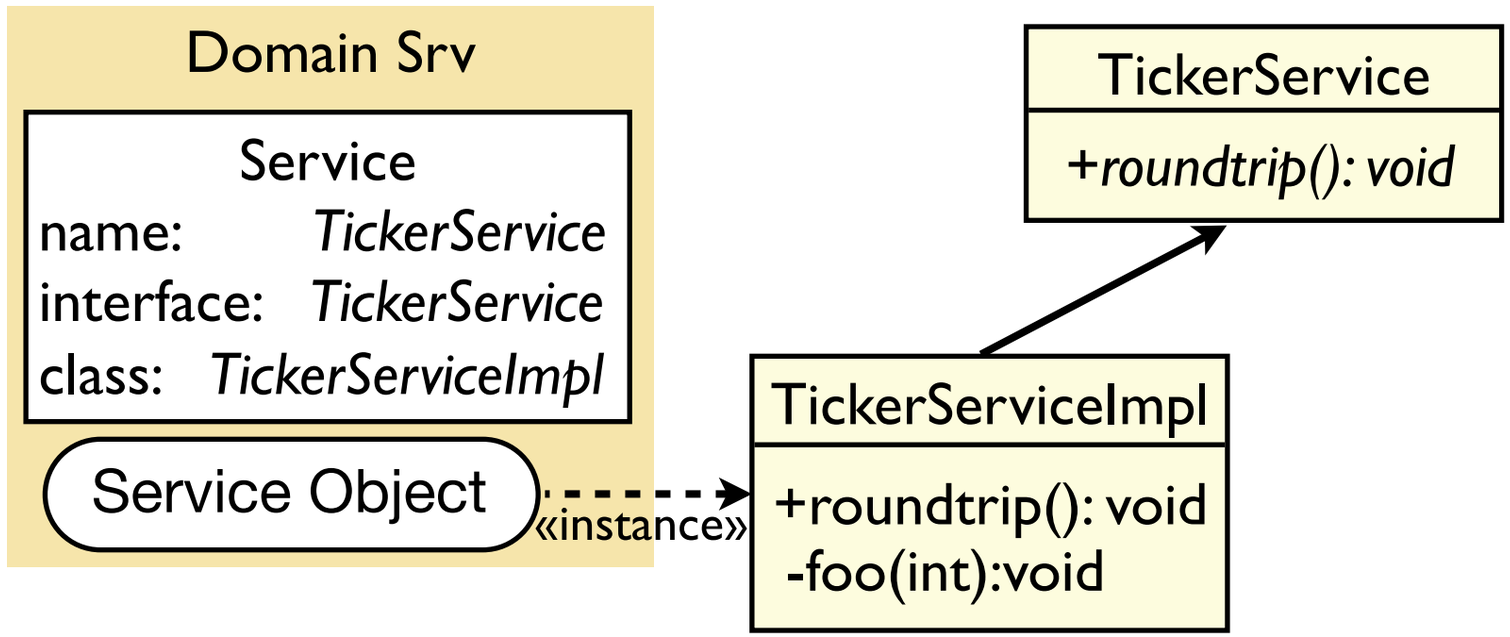
Inter-Domain Communication with Portals

- service domain: exports interface as named service



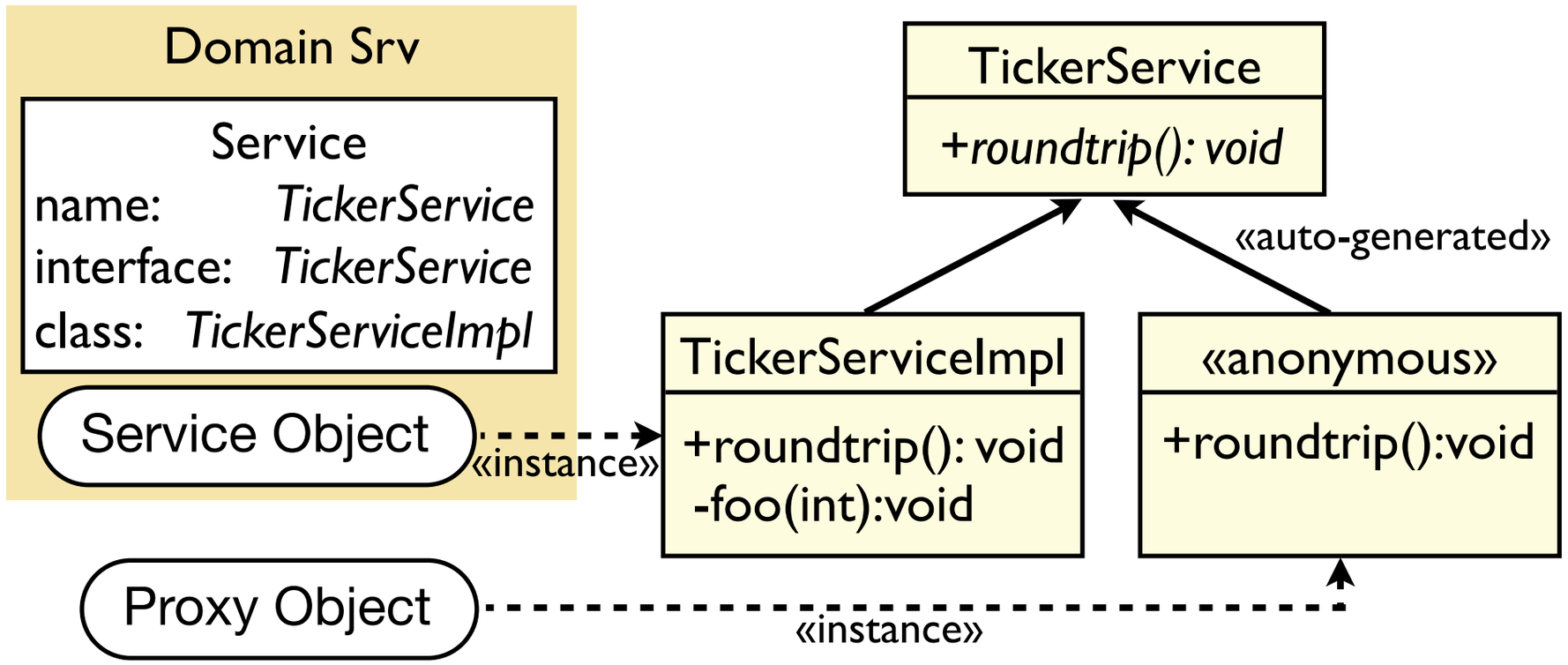
Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain



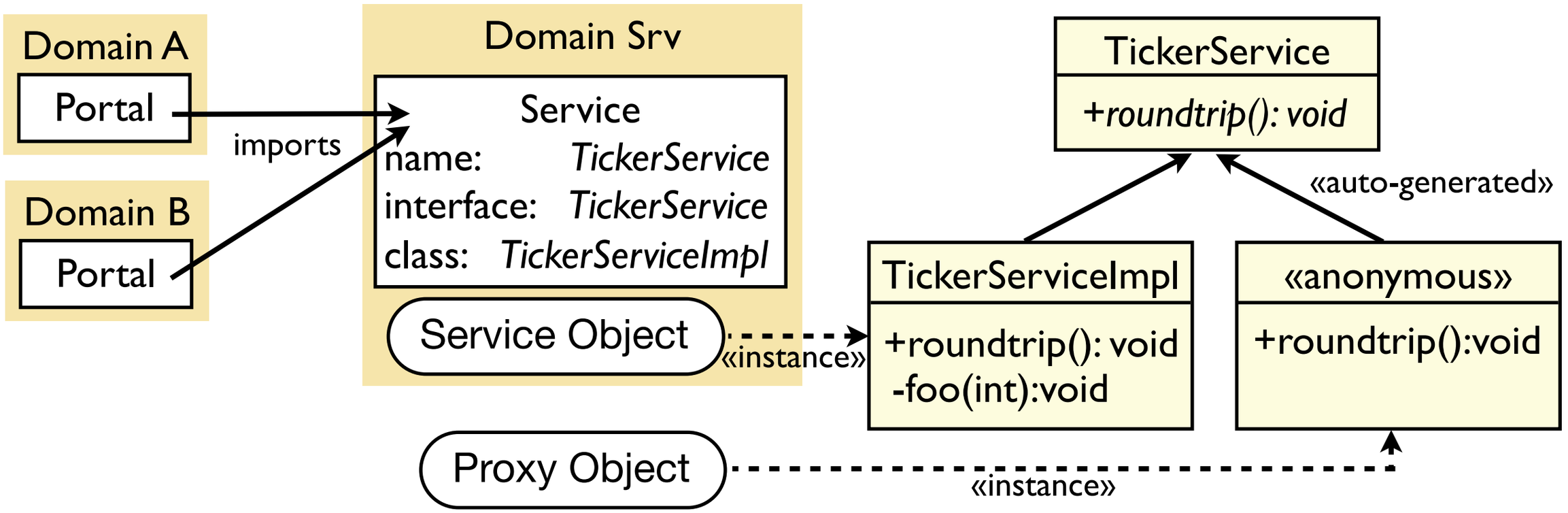
Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains



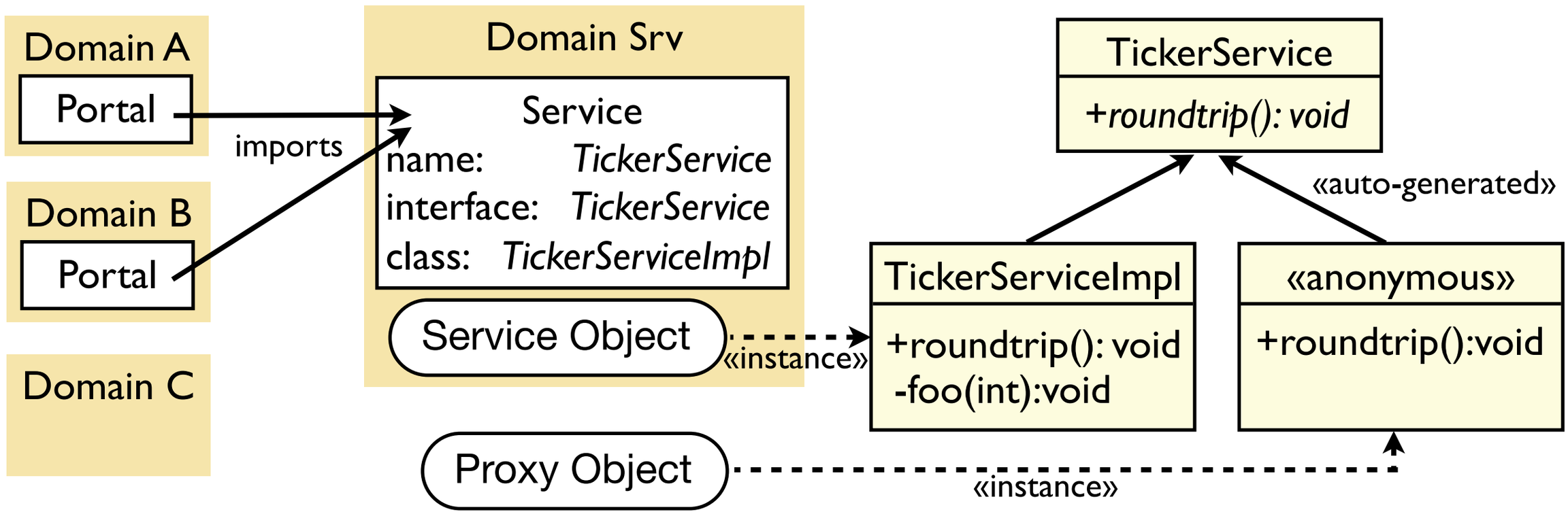
Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains
- client domains: statically import this service
  - client domains acquire proxy object reference via name service



Inter-Domain Communication with Portals

- service domain: exports interface as named service
  - service object is allocated in the service domain
  - proxy object (portal) is statically allocated for the client domains
- client domains: statically import this service
  - client domains acquire proxy object reference via name service
  - other domains cannot access the service at runtime



Portals: Parameter Passing

- strictly call-by-value
  - retain logical heap separation
- reference parameters
  - deep copy to the service domain's heap
  - GC needed in the service domain
- marker interface `NonCopyable` to prevent copying
  - reference replaced by `null`
  - used for system objects

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup(”TickerService”);

  srv.roundtrip();
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup(”TickerService”);

  srv.roundtrip();
}
```

Name Service

- compiled to an array lookup
- returns
  - service object in service domain
  - portal object in client domains
  - `null` otherwise

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup("TickerService");

  srv.roundtrip();
}
```

Portal Call

- regular virtual method call

Portals: Implementation and Overhead

```
public void foo() {
   TickerService srv = (Tick
         PortalService.looku

   srv.roundtrip();
}
```

Switch Protection Context

- backup current domain on stack
- change current execution context
- migrate task to service domain
- restore original domain on return

```
public void roundtrip_portal(object_t *proxy) {
   domain_t prev_domain = CURRENT_DOMAIN;
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService

  srv.roundtrip();
}
```

Partition Stack

- enables GC to skip irrelevant partitions
- only if service method potentially blocks

```
public void roundtrip_portal(object_t *proxy) {
   domain_t prev_domain = CURRENT_DOMAIN;
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService sr
        PortalServ

  srv.roundtrip();
}
```

Invoke Service Method

- statically bound call
- service object is passed as `this` reference
- primitive parameters are passed through
- references are deep copied

```
public void roundtrip_portal(object_t *proxy) {
   domain_t prev_domain = CURRENT_DOMAIN;
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Portals: Implementation and Overhead

```
public void foo() {
  TickerService srv = (TickerService)
        PortalService.lookup(”TickerService”);

  srv.roundtrip();
}
```

```
public vo
   domain_
   CURRENT_TASK->effective_domain = DomainSrv_ID;
   CURRENT_DOMAIN = DomainSrv_ID;
   PUSH_STACK_PARTITION(DomainSrv_ID);
   roundtrip_impl(&tickerservice_srvobj);
   POP_STACK_PARTITION();
   CURRENT_DOMAIN = prev_domain;
   CURRENT_TASK->effective_domain = prev_domain;
}
```

Cost

- primitive parameters: same order of magnitude
- reference parameters: next order(s) of magnitude

Domains compared to Java Isolates (JSR-121)

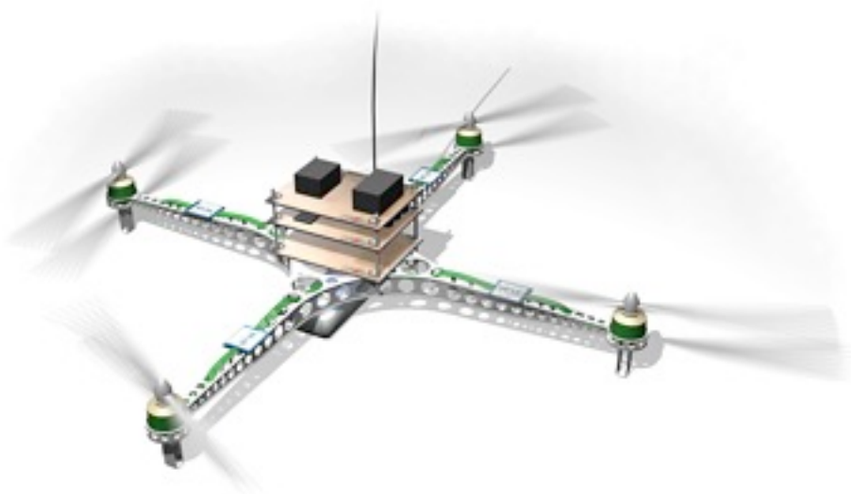
	Domains	Isolates
isolation concept	realm-local static fields logical heap separation	
instantiation	ahead-of-time	at runtime
runtime representation	none (to the application)	instances isolate status lifecycle messages
communication	portals shared memory	message links inter-JVM mechanisms

Performance: KESO vs. C

- Flight Attitude Control Algorithm of the I4Copter
  - http://www4.cs.fau.de/Research/I4Copter
  - C-Code generated from a Simulink Model
  - Input: sensor and steering data
  - Output: engine thrust levels
  - period 9ms
  - Tricore TC1796b MCU @150 MHz (1 MiB MRAM)
- Java port close to the C version
- Recorded trace of inflight sensor and steering data
  - Verified that C and Java version output the same actuator values
  - Replayed 200 data samples to measure execution time

Performance: KESO vs. C

- Flight Attitude Control Algorithm of the I4Copter
  - http://www4.cs.fau.de/Research/I4Copter
  - C-Code generated from a Simulink Model
  - Input: sensor and steering data
  - Output: engine thrust levels
  - period 9ms

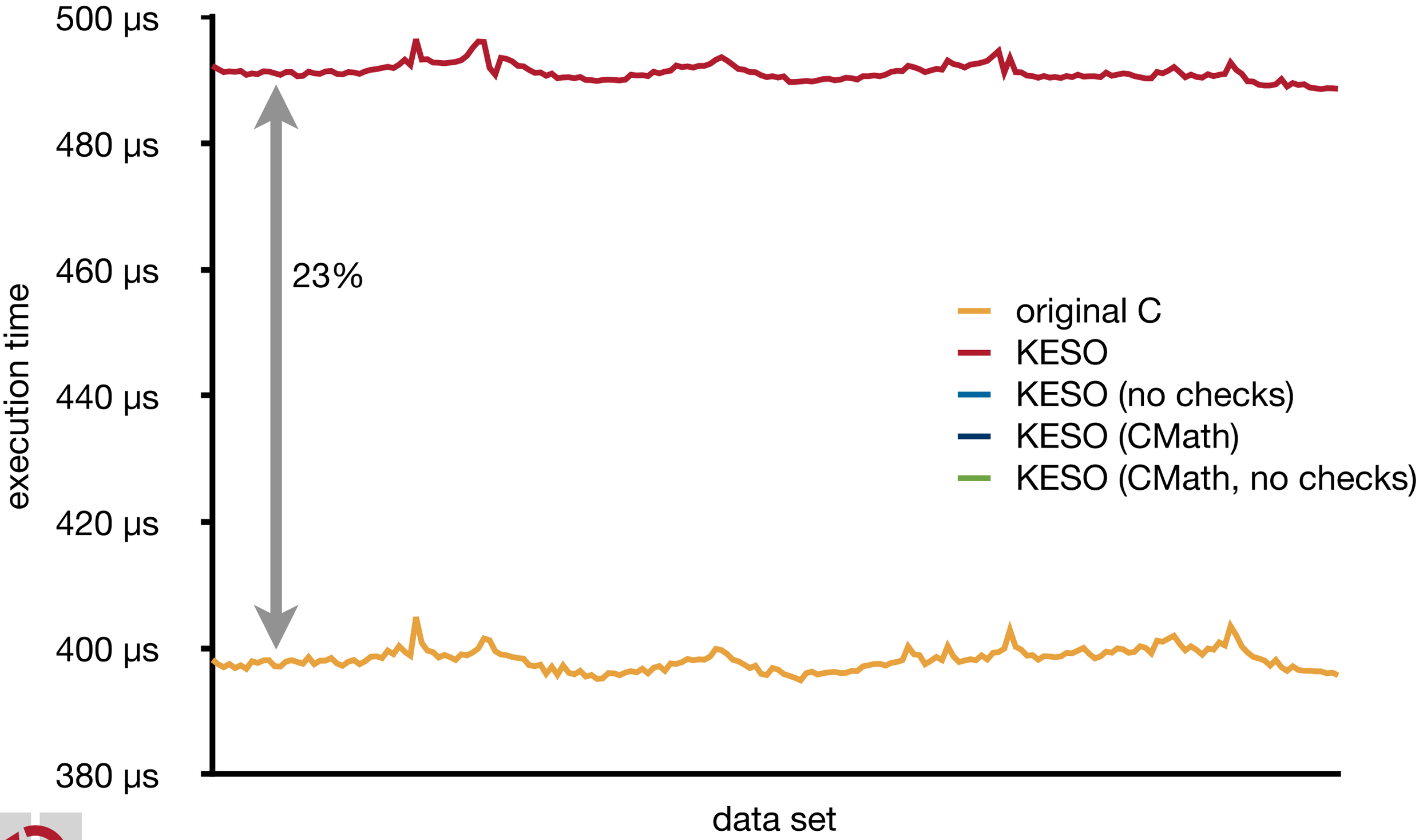


  - Tricore TC1796b MCU @150 MHz (1 MiB MRAM)

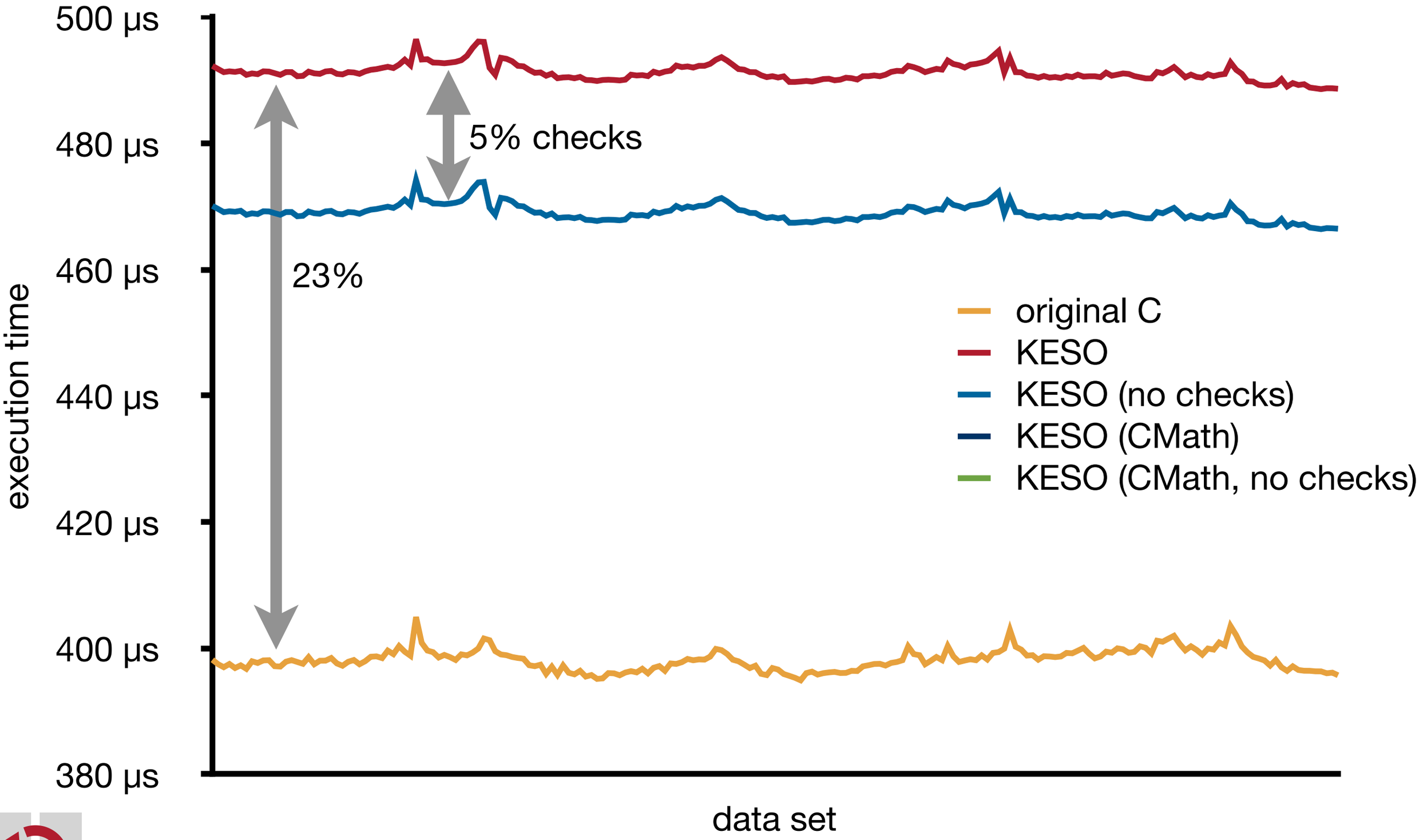
- Java port close to the C version

- Recorded trace of inflight sensor and steering data
  - Verified that C and Java version output the same actuator values
  - Replayed 200 data samples to measure execution time

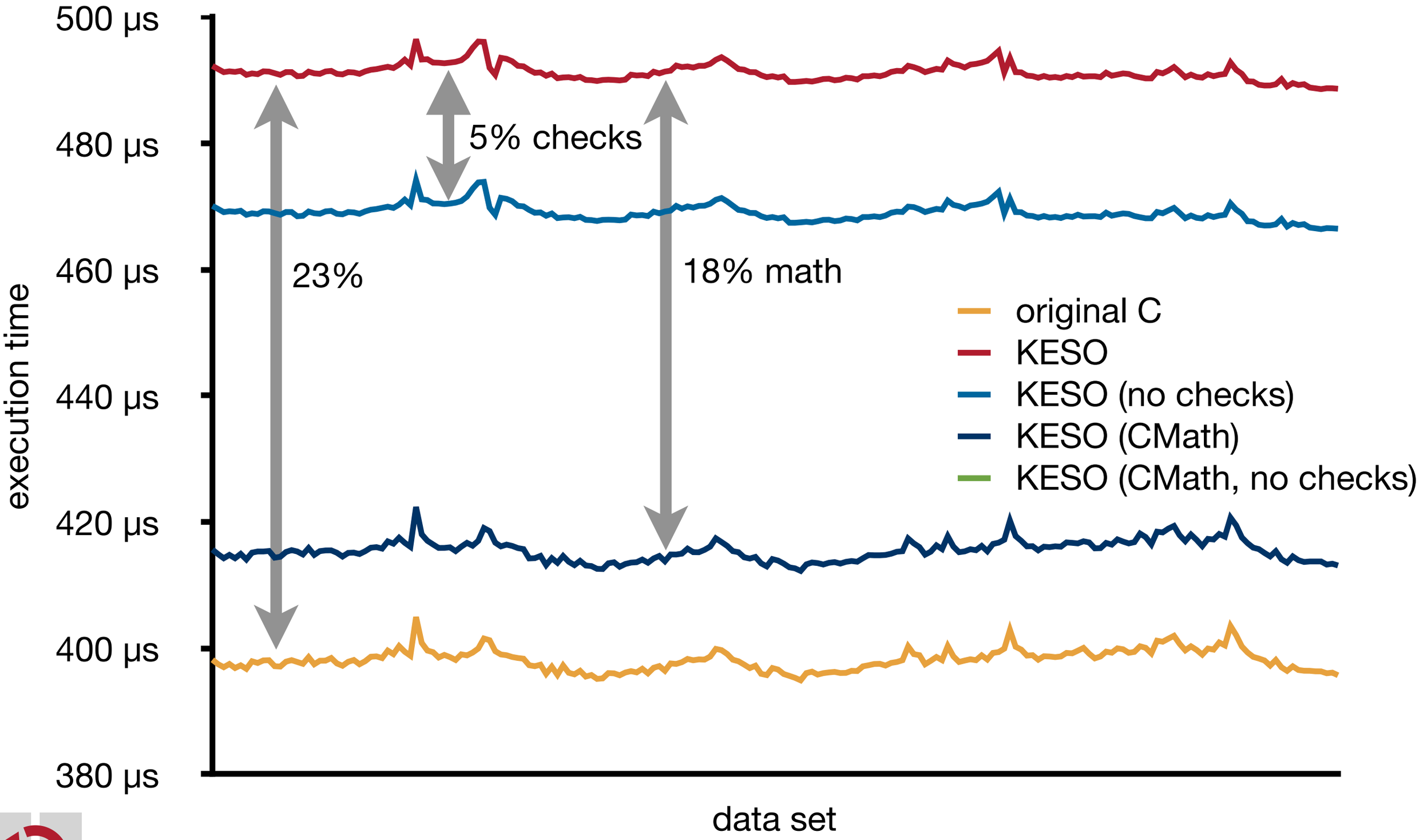
Attitude Controller - Execution Times



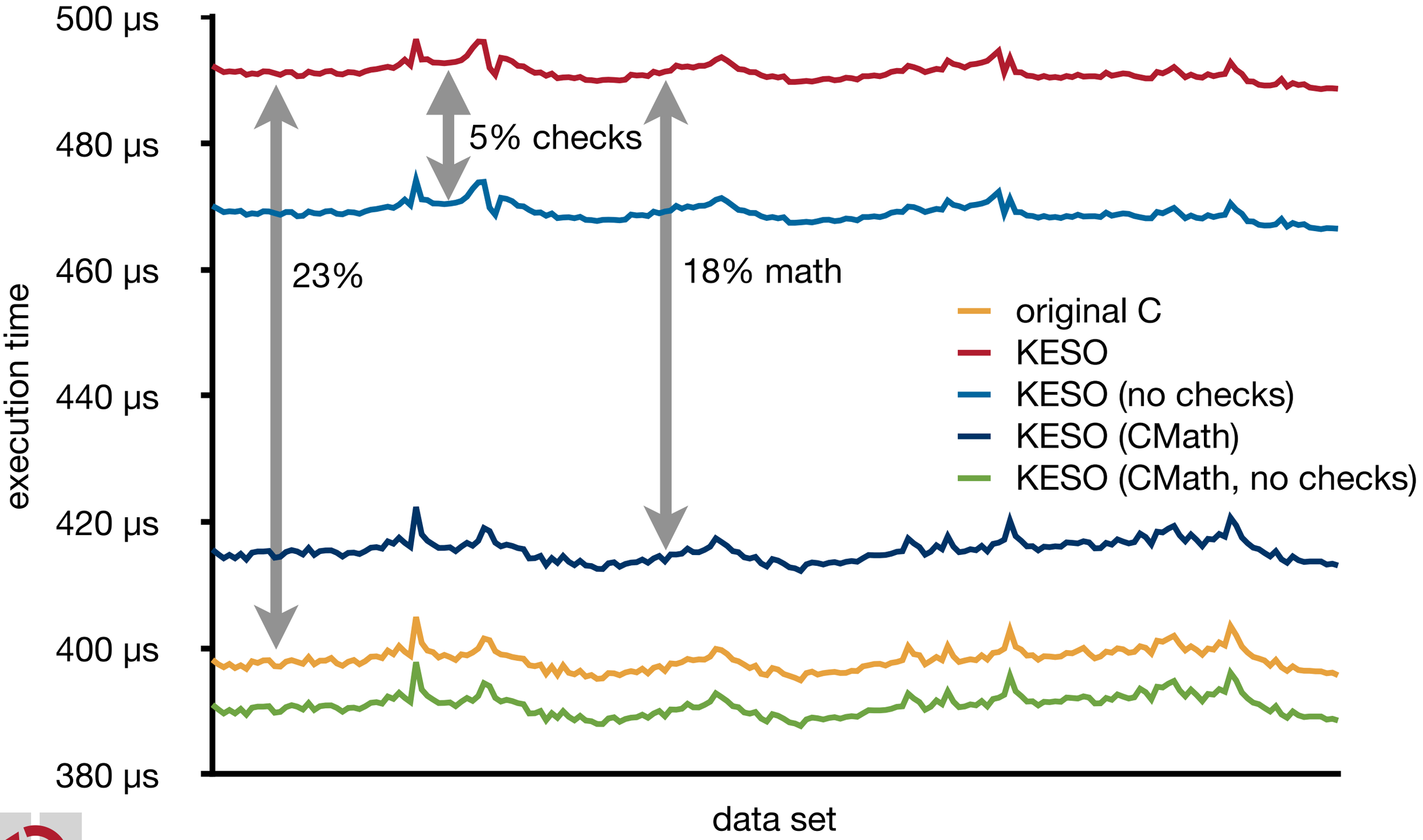
Attitude Controller - Execution Times



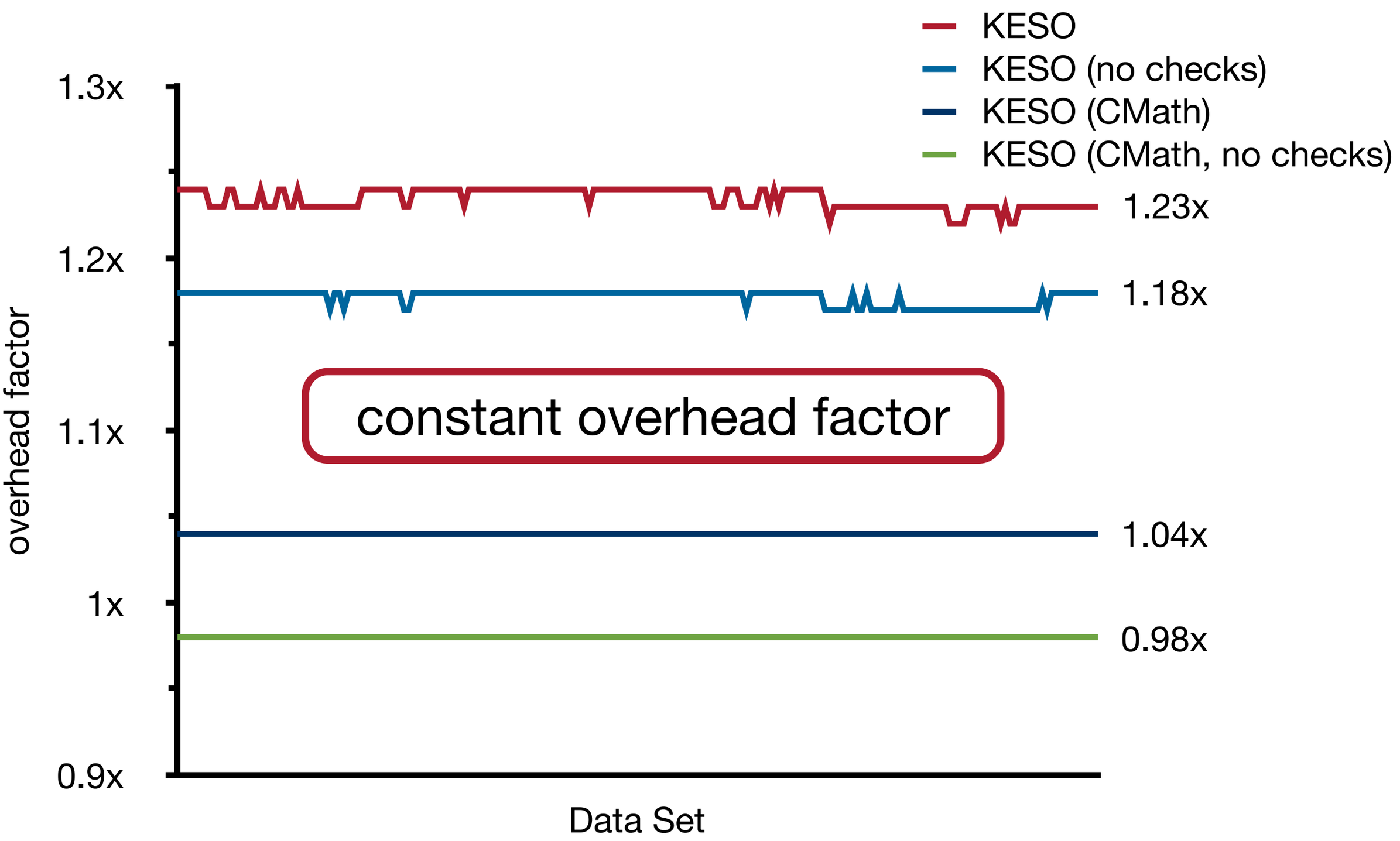
Attitude Controller - Execution Times



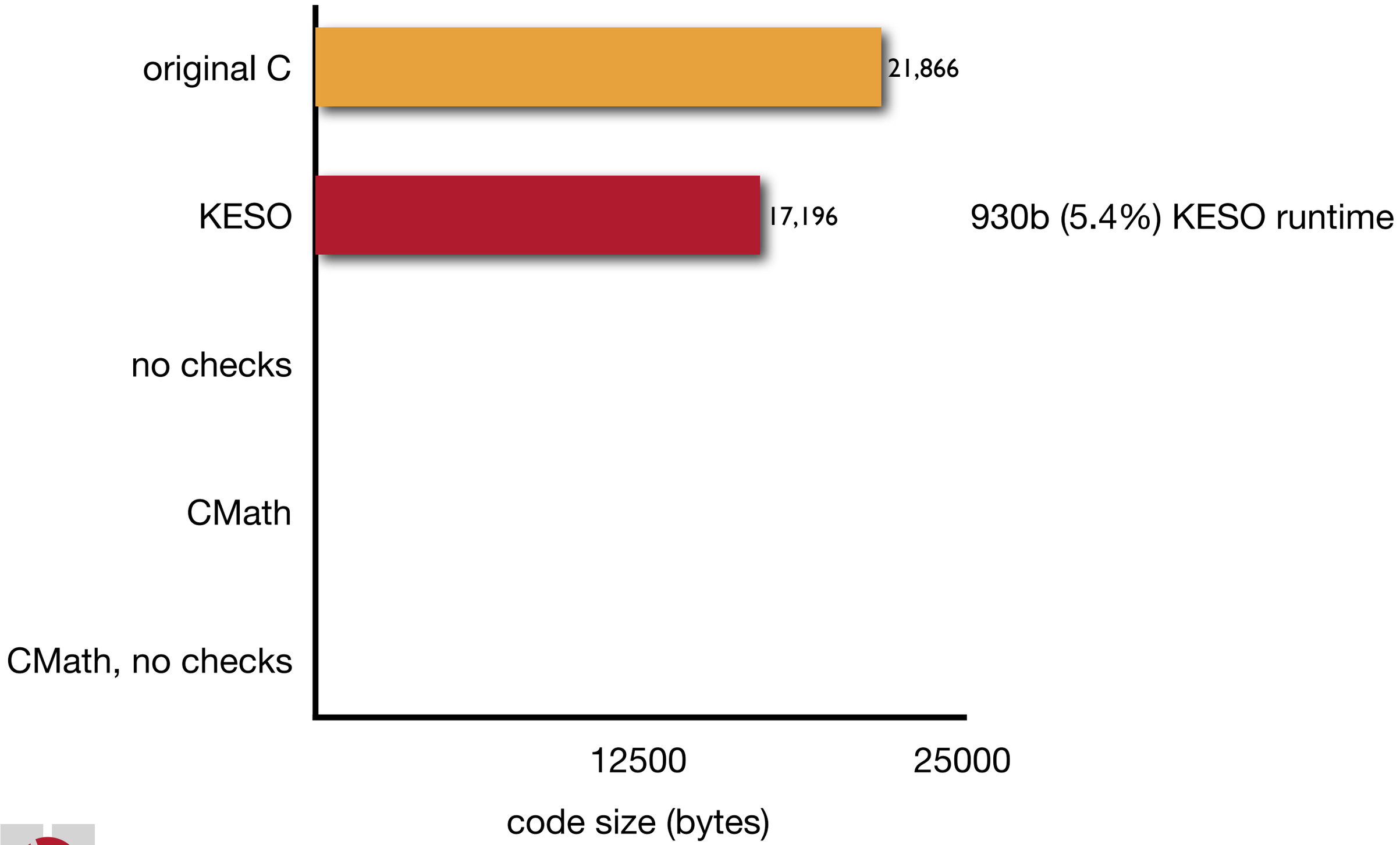
Attitude Controller - Execution Times



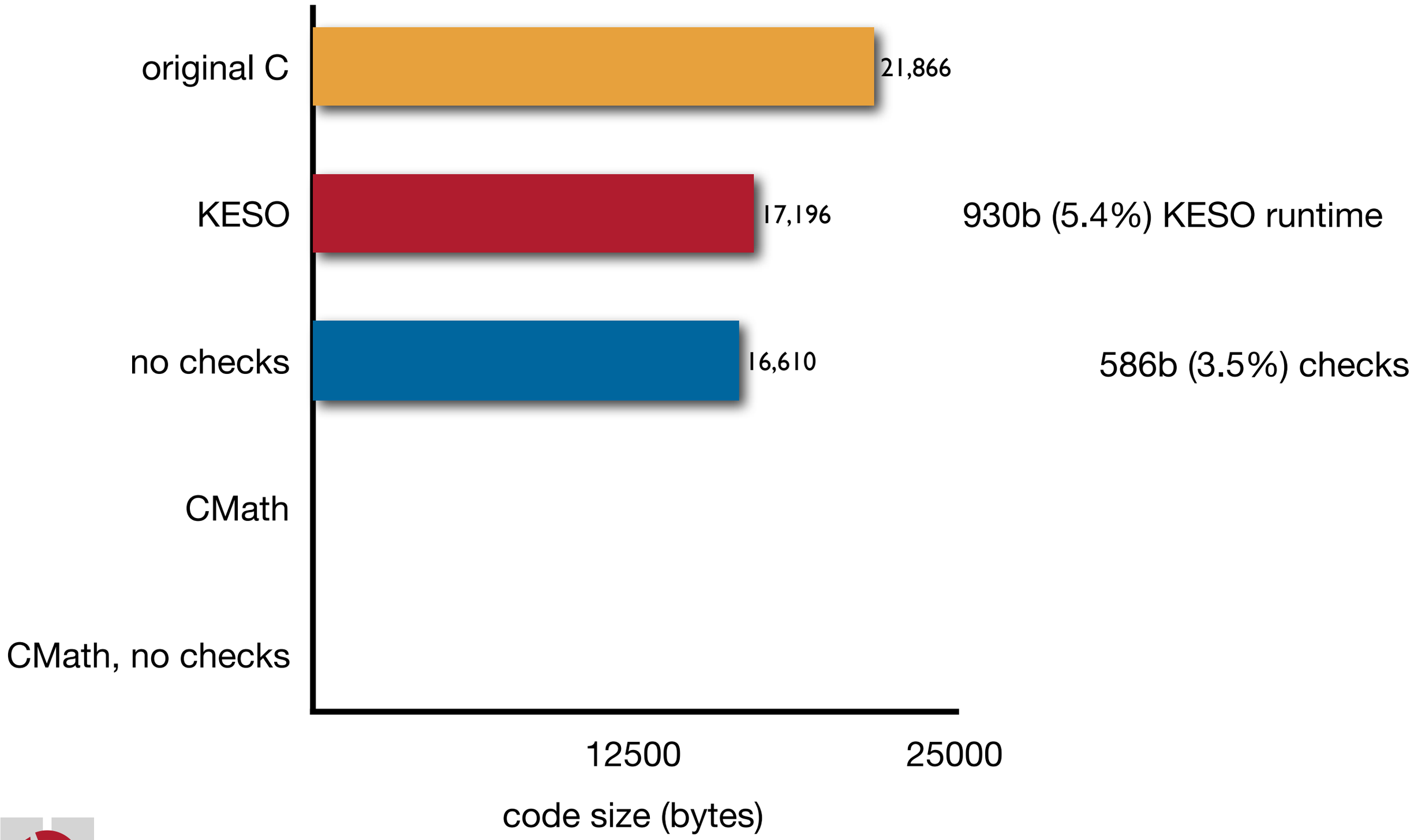
Attitude Controller - Execution Times



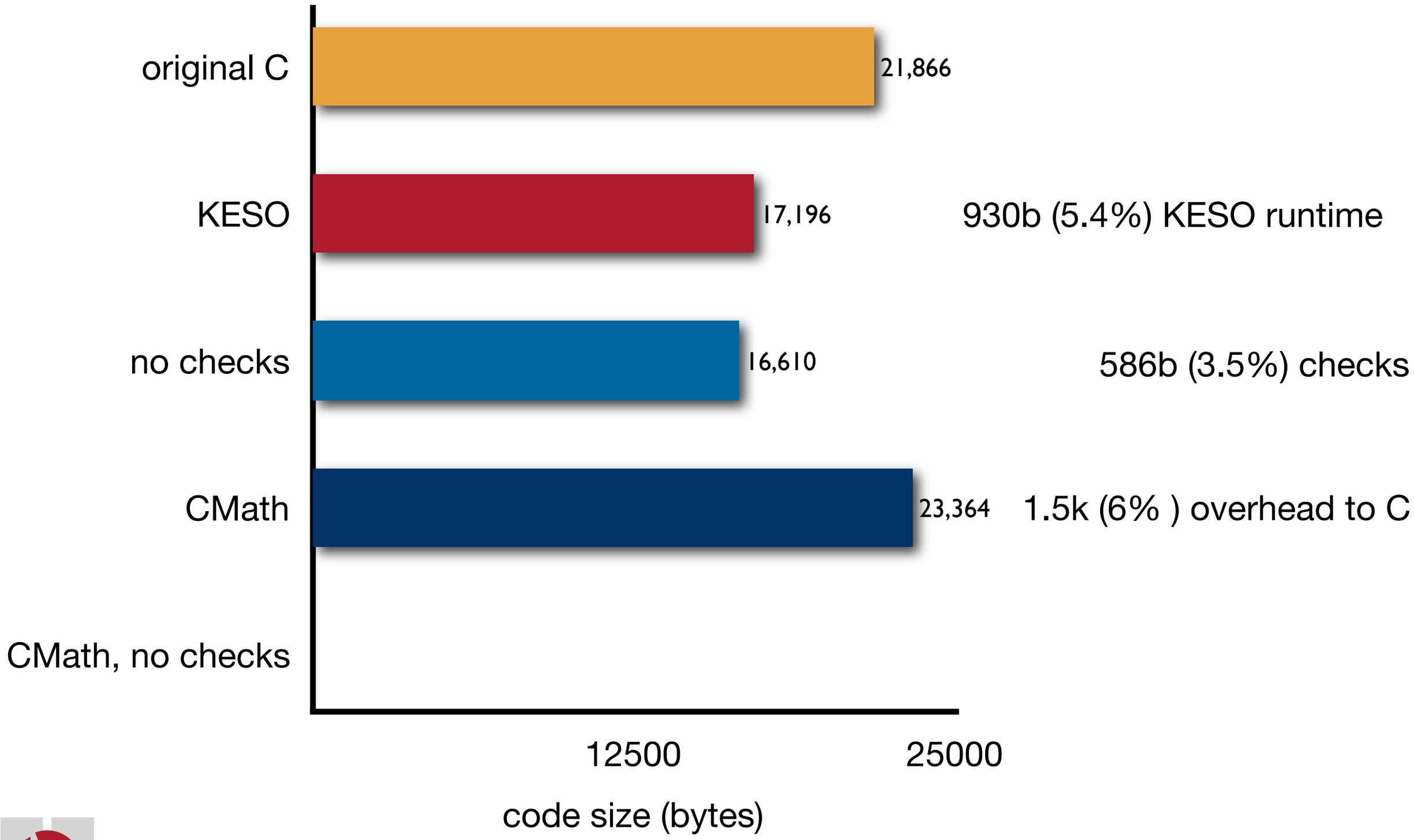
Attitude Controller - Code Size



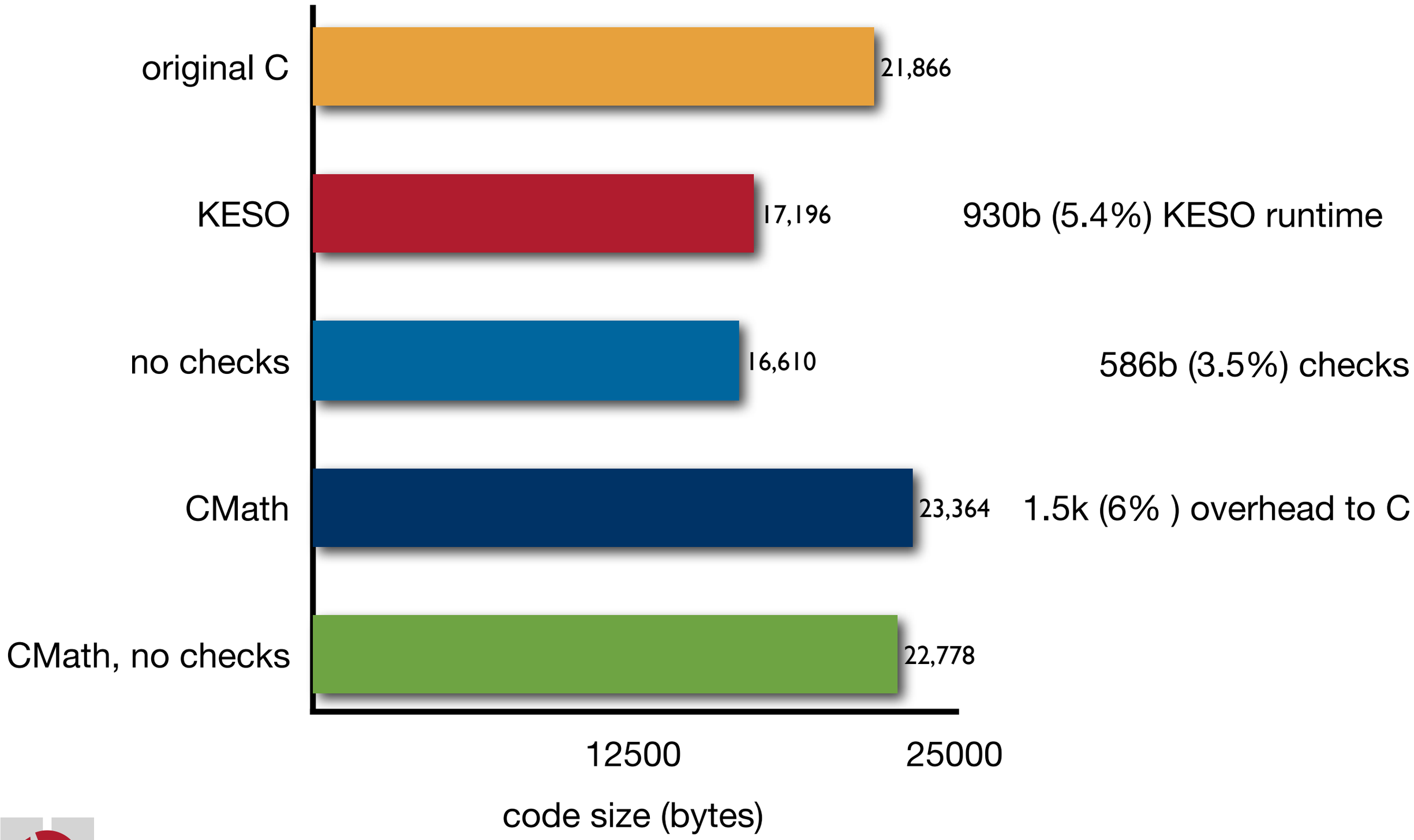
Attitude Controller - Code Size



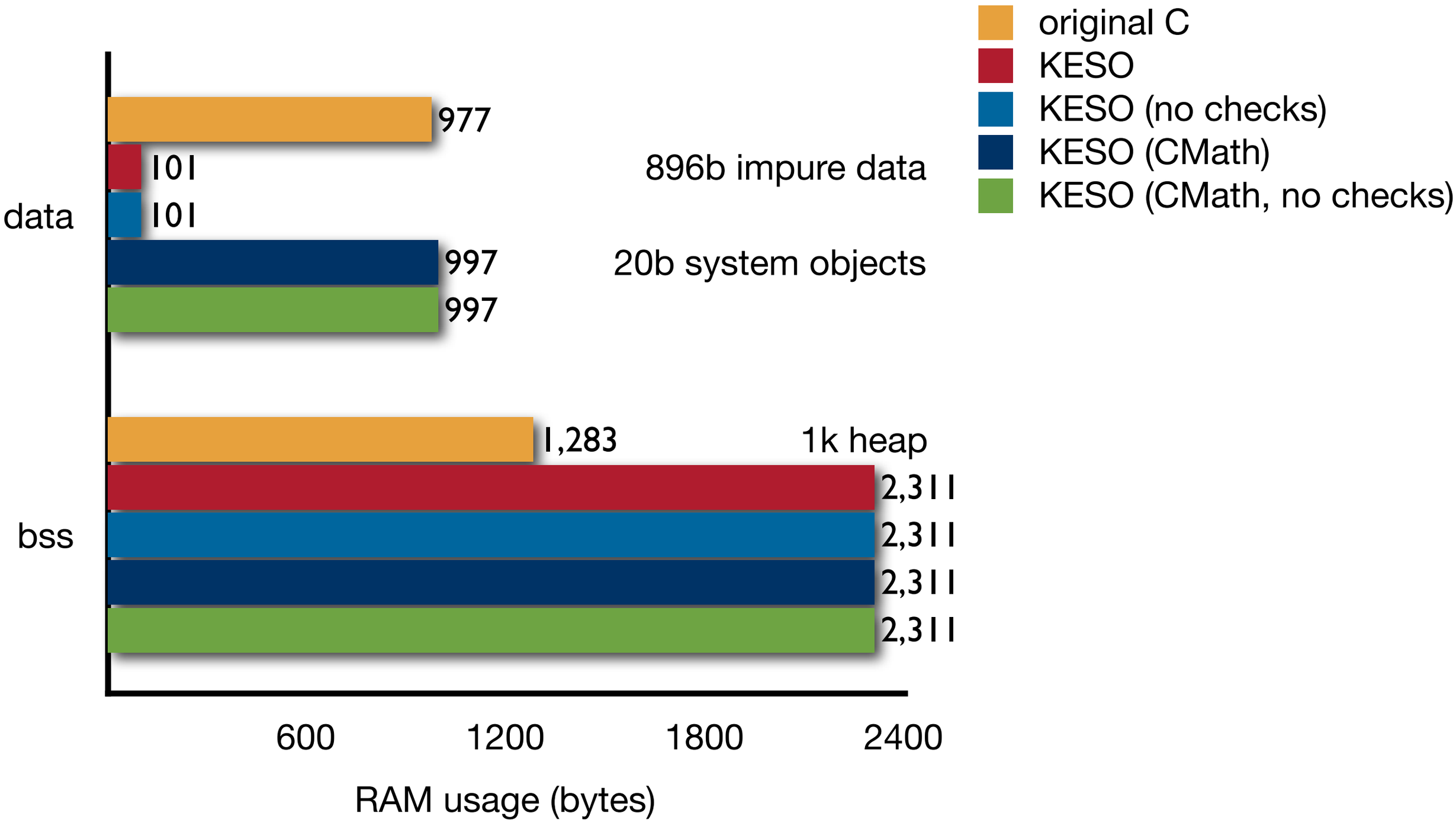
Attitude Controller - Code Size



Attitude Controller - Code Size



Attitude Controller - RAM Usage



Conclusions

- KESO targets the domain of static embedded systems

- VM tailoring
  - small overhead of the runtime environment
  - few fixed restrictions to the VM feature set
  - you pay what you need/use

- performance and footprint competitive to C
  - ahead-of-time compilation
  - static analyses

- software-based spatial isolation