The Open Kernel Environment

(opening up all levels of the processing hierarchy in a 'safe' manner)

Herbert Bos
Bart Samwel

Leiden University
{herbertb,bsamwel}@liacs.nl

http://www.liacs.nl/~herbertb/projects/oke/

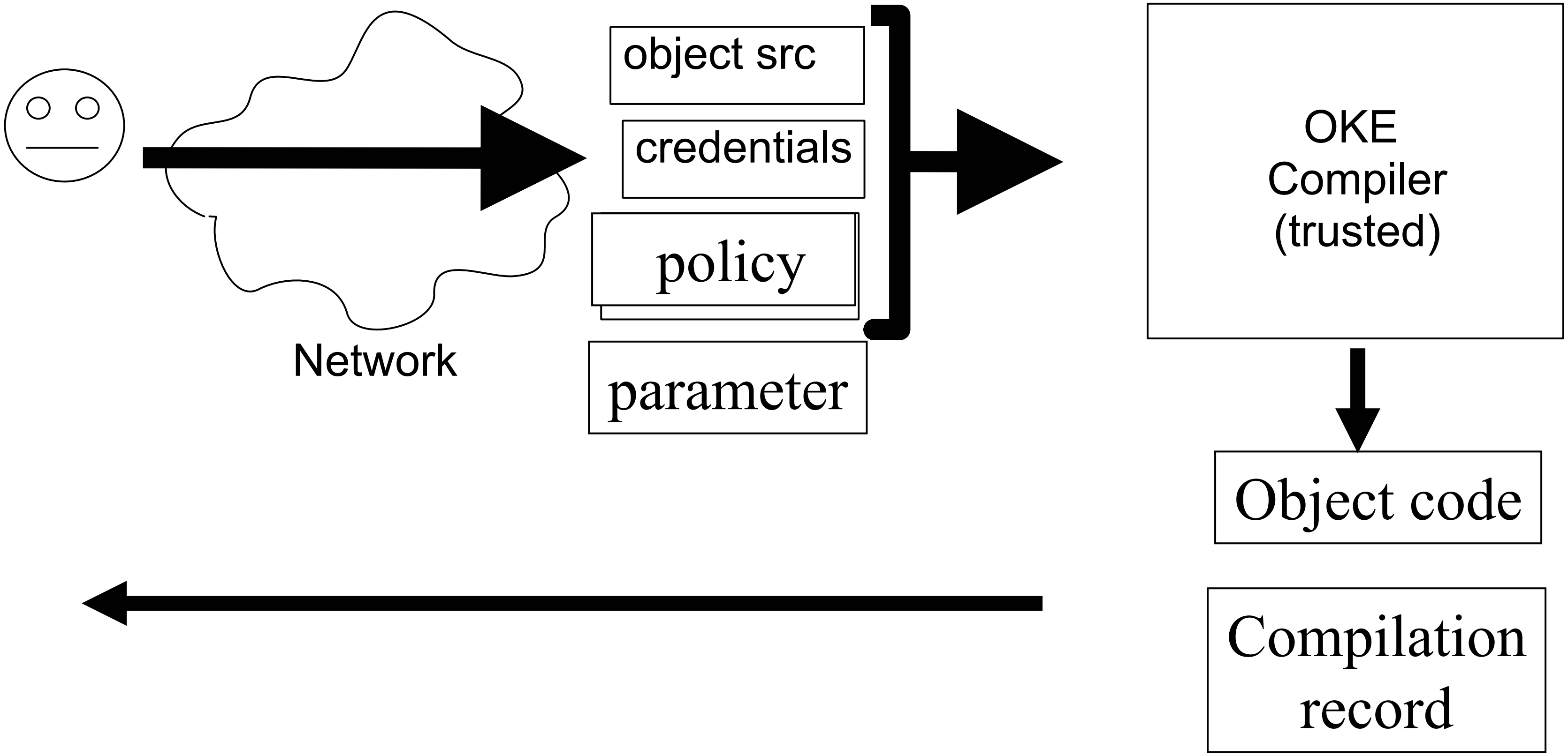
What is this about?

- goals:
  (1) allow 3rd party programming of lower levels
  (2) safety enforced by software based isolation
- target:
  environments with little hardware support for isolating applications
  - so, no MMU or privileged instructions
  - example: Linux kernel, network processors
- fully optimised native code / speed
- safety/resource control
  - CPU, heap, stack, API, etc

OKE: Open Kernel Environment

- *explicit trust management + trusted compiler*
- compiler: restrict code's access to resources depending on privileges
- code loader
- ***accepts code + authentication + credentials***
- ***if match : load code of specific type***
- ***type = with specific resource restrictions***
  - ***instantiation of type happens by parameterisation***
  - ***example: type = { MAX_HEAP < 1 MB }***
  - ***example: instantiation = { MAX_HEAP = 100 kB }***
- restrictions
- ***CPU, heap, stack (recursion), APIs***
- ***pointer, private data, access to memory bus (and more)***

OKE Compiler



Network

object src

credentials

policy

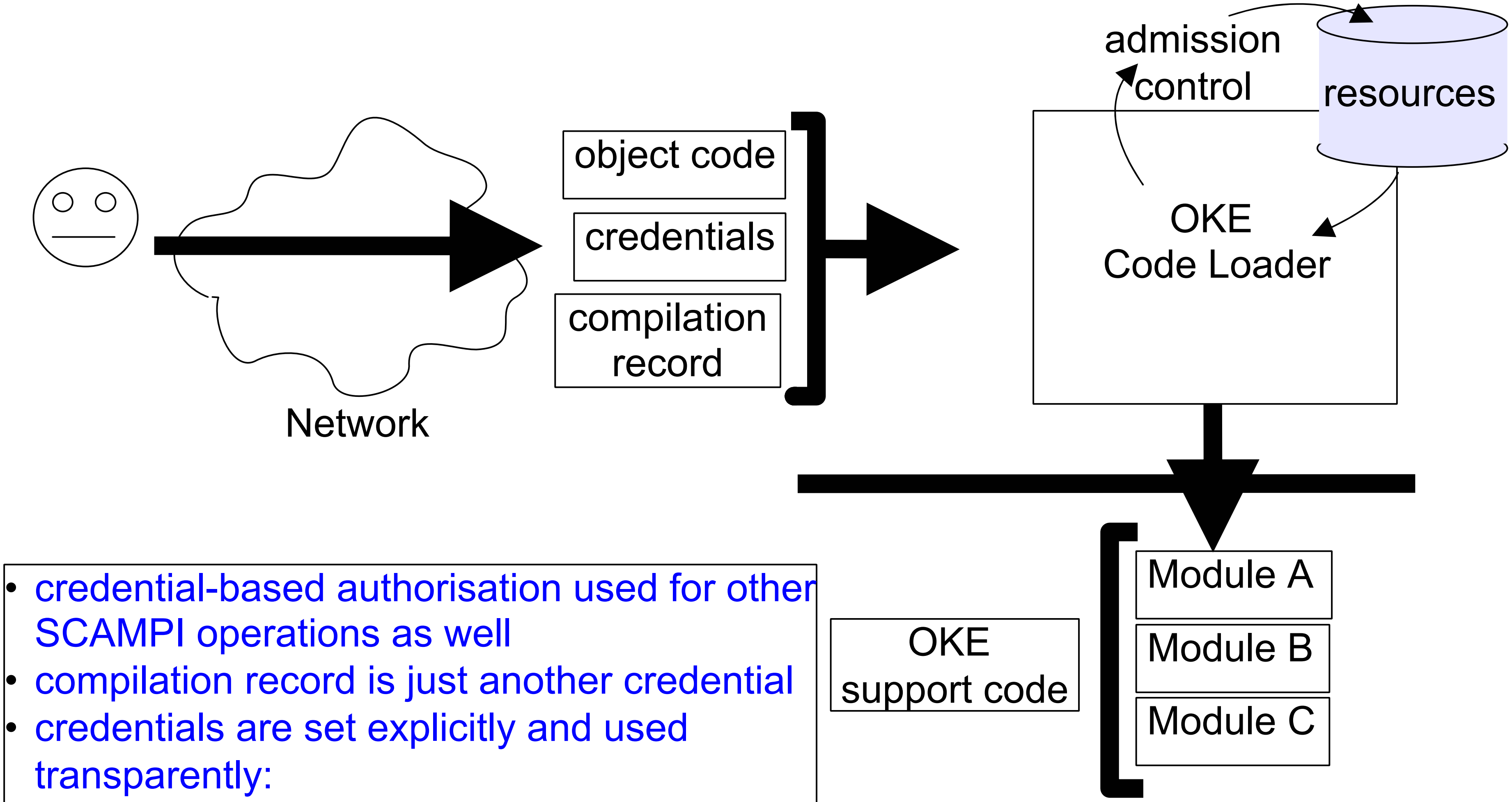
parameter

OKE
Compiler
(trusted)

Object code

Compilation record

OKE Code Loader



Network

object code

credentials

compilation record

admission control

resources

OKE Code Loader

OKE support code

Module A

Module B

Module C

- credential-based authorisation used for other SCAMPI operations as well
- compilation record is just another credential
- credentials are set explicitly and used transparently: `scampi_set_authorisation_cred ()`

OKE Credentials: delegated trust management

- delegated trust management using KeyNote and OpenSSL
- condition can be environment specific

```
KeyNote-Version: 2
Comment: trivial policy: authorise licensees for operation 'createFlow'
Authorizer: "POLICY"
Licensees: "rsa-base64:MEgCQQDMcZukqn3Wa4Z2y3wKljB/eoFnDRfNN\
            B72OJLsfW6SnFRLKbXrgEnEP+7LevQEI0KsUq8NsgQmtx1btq\
            lqyETdAgMBAAE="
Conditions: app_domain == "SCAMPI.MAPI" && op == "createFlow" -> "true";
```

```
KeyNote-Version: 2
Comment: OKE CL credential authorising client to load code of this type
Authorizer: "rsa-base64:MEgCQQDMcZukqn3Wa4Z2y3wKljB/eoFnDRfNN\
            B72OJLsfW6SnFRLKbXrgEnEP+7LevQEI0KsUq8NsgQmtx1btq\
            lqyETdAgMBAAE="
Licensees: "rsa-base64:ABCDE12345"
Conditions: app_domain == "SCAMPI.MAPI" && op == "createFlow" \
            && param2 == "10.0.0.1" -> "true";
Signature: "sig-rsa-sha1-base64:Du1uNVtNv8sAhjni/8UnzI9H+/VM\
            9GnSM/ppgfEOAmO/QzSESYZgrwsMEPlzAFqnbNGfwusxlXEIz\
```

Environment Setup Code (ESC)

- policies implemented as ESC that is automatically prepended to user code
  - defines runtime support
  - explicitly declares API the code can use
  - removes the ability to
    - declare/import new APIs
    - access the 'private parts' of ESC
    - perform unsafe operations
- 1 translation unit => whole program analysis

  - part 1: macro definitions expressing parameterisation
  - part 2: ESC
  - part 3: user code

OKE programming language

- many different user 'classes' with different trade-offs regarding the amount of restriction needed
  - students
  - system administrators
  - anonymous
- avoid special-purpose languages (rather: 1 language that can be customised)
- interfacing with rest of kernel important
- ideally something like C
- Cyclone ("a crash-free dialect of C")

Language features for safety

- Cyclone: strongly typed, pointer protected, garbage collected, and provides region-based mem protection
- A measure of safety is provided by combination of existing Cyclone and new features
- Spatial pointer safety
  - bounded pointers
  - non-nullable pointers -> not checked
  - 'normal' C pointers -> always checked

Language features for safety

- Temporal pointer safety
  - region-based protection (e.g. to prevent returning the address of a local variable), added kernel region
  - RBP and GC work well for Cyclone-only, but present safety issues when interacting with C code
  - for example: suppose an OKE module holds a pointer to kernel memory
  - OKE solution:
    - delayed freeing plus (place blocks on kill list)
    - new GC: O(n) in # of allocated blocks)
    - GC round just prior to activation of the OKE module (hmmm...)
    - nullifies pointers or kills modules
    - delayed freeing: not always needed

Language features for safety

- Language restrictions
  forbid construct, e.g.:
  - forbid extern "C" // no import of C APIs
  - forbid namespace ... // no access to specific namespace
  - forbid catch ... // do not catch certain exceptions
- API and entry point wrapping
  - potential entry points explicitly declared
    (pointers can only be taken of functions declared extern)
  - automatically wrapped in ESC ("wrap extern")
  - kernel APIs may also be wrapped

Language features for safety

- Sensitive data protection at compile time
  locked construct
  - sharing: parts of a data structure should be inaccessible
  - normal solution: anonymising
  - locked variables cannot be used in calculations and cannot be cast
  - may be declared const

Language features for safety

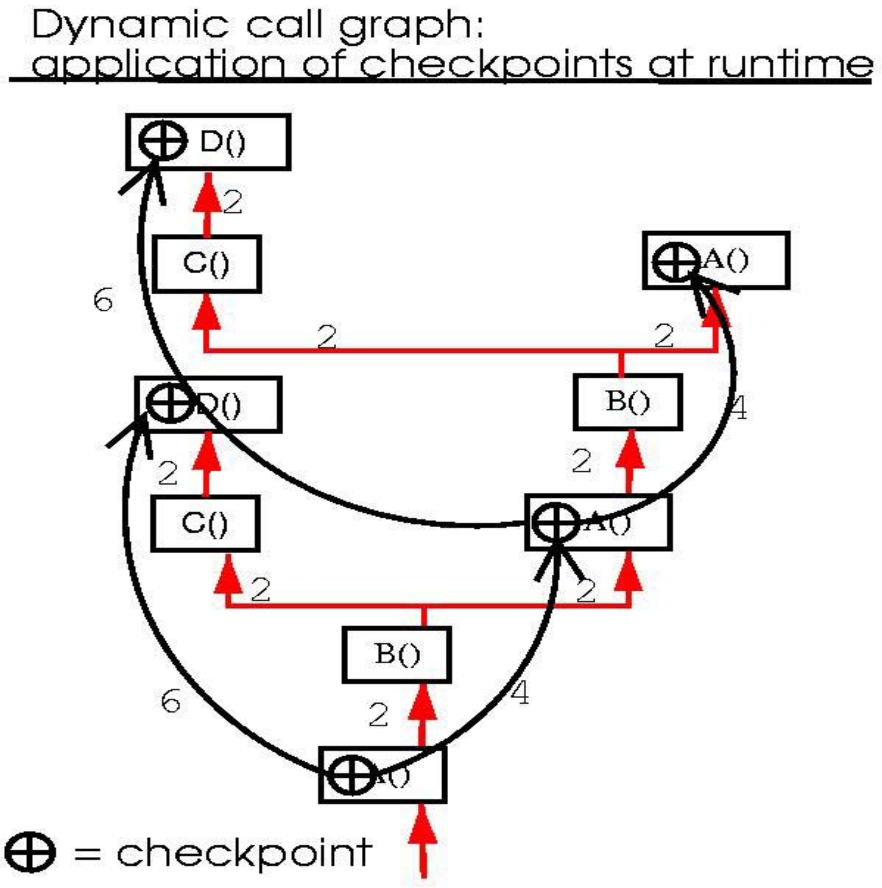
- Stack overrun protection
  - some dynamic checking needed (but flexible)
  - 2 parameters: bound and granularity

```
int D() {
  return 0;
}

int C() {
  return D();
}

int B() {
  if (cond)
     return C();
  else return A();
}

int A() {
  return B();
}
```

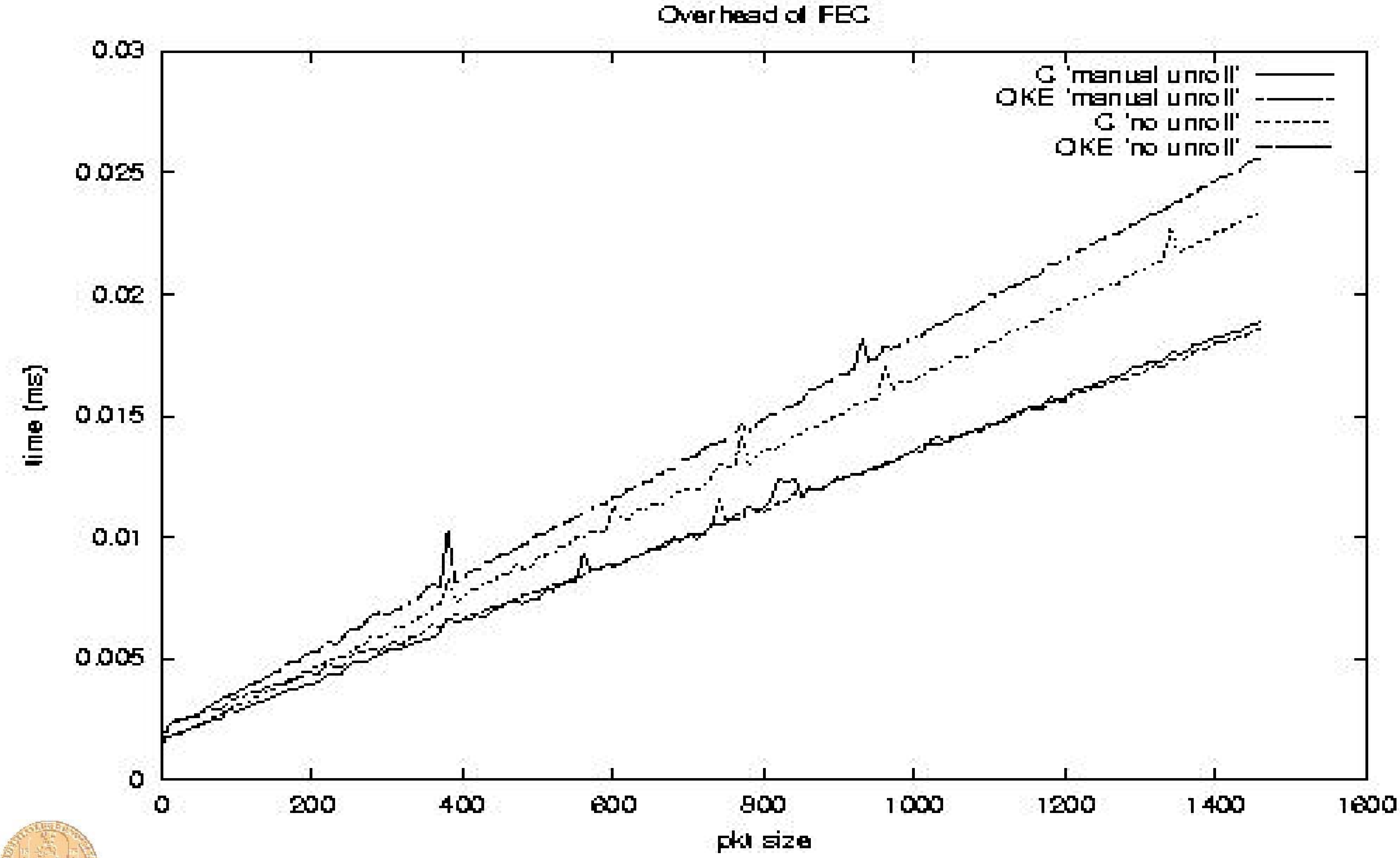


Dynamic call graph:
application of checkpoints at runtime

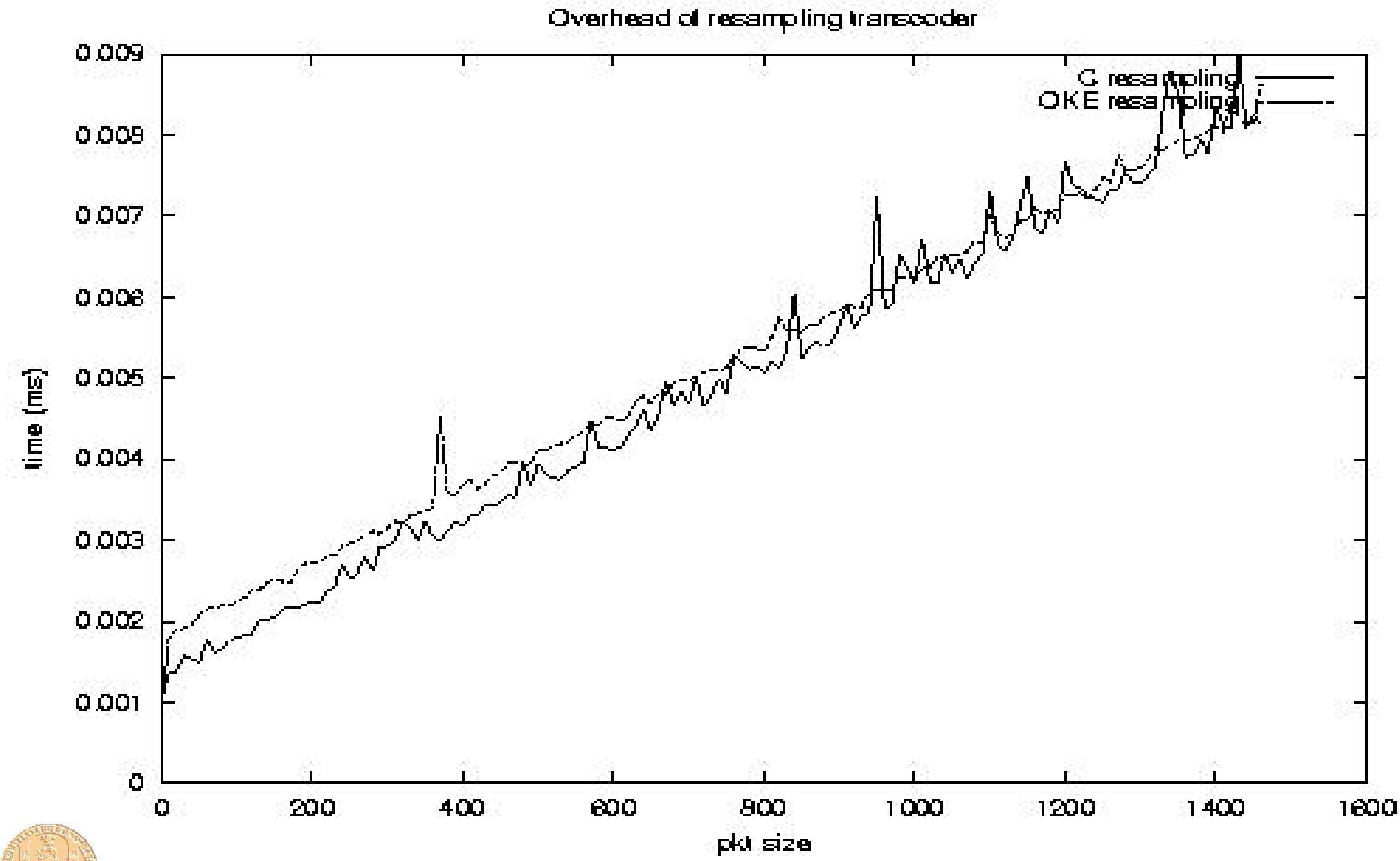
OKE features for safety

- timeout protection
  - multiple solutions
    - dynamic checks in backward jumps
    - timer interrupt: applied in Linux kernel
    - other: applied in IXP1200 network processor
  - timer interrupt
    - on return from interrupt: check if timeout is detected
    - if so, jump to callback function registered by ESC
    - this function may throw exception
    - takes into account if code is executing kernel or OKE code

FEC overhead over C



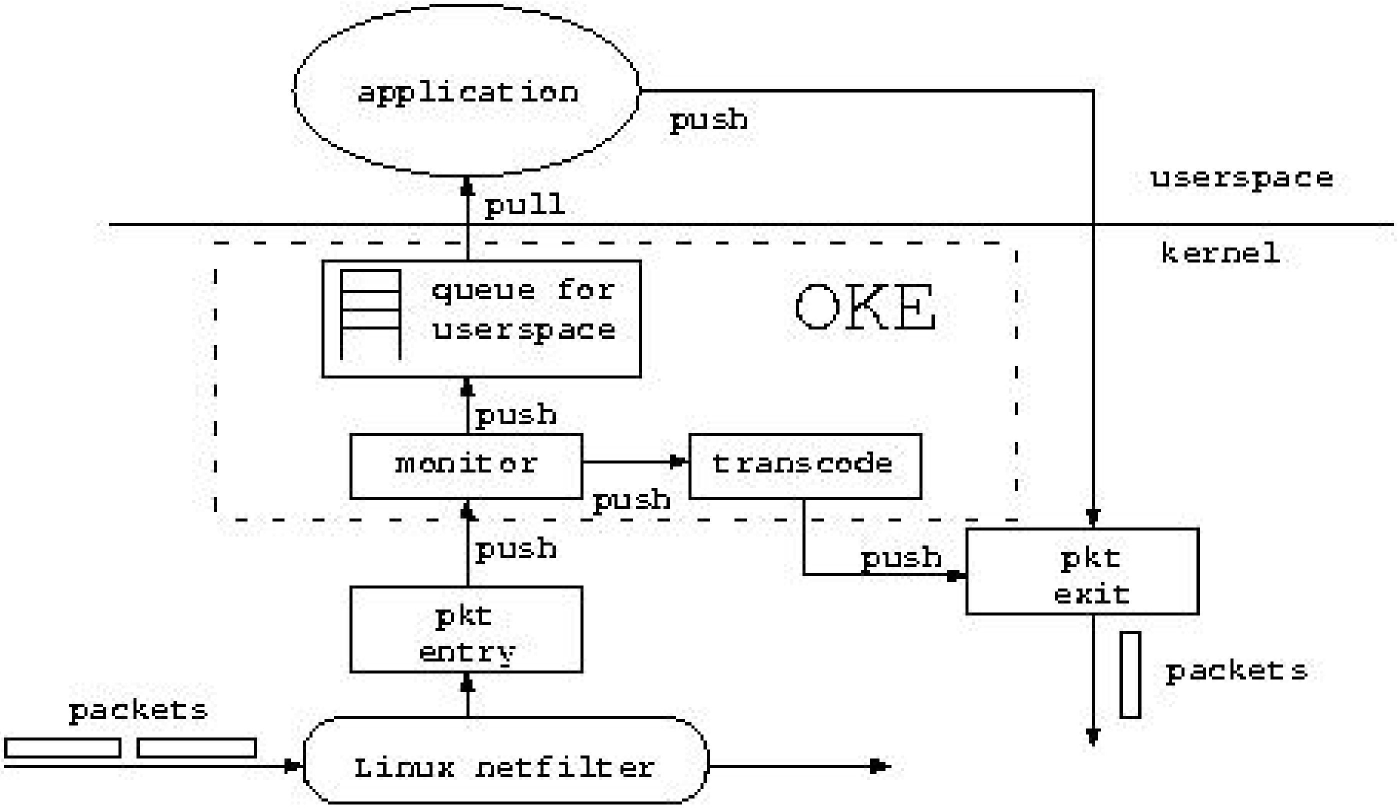
Audio resampling overhead over C



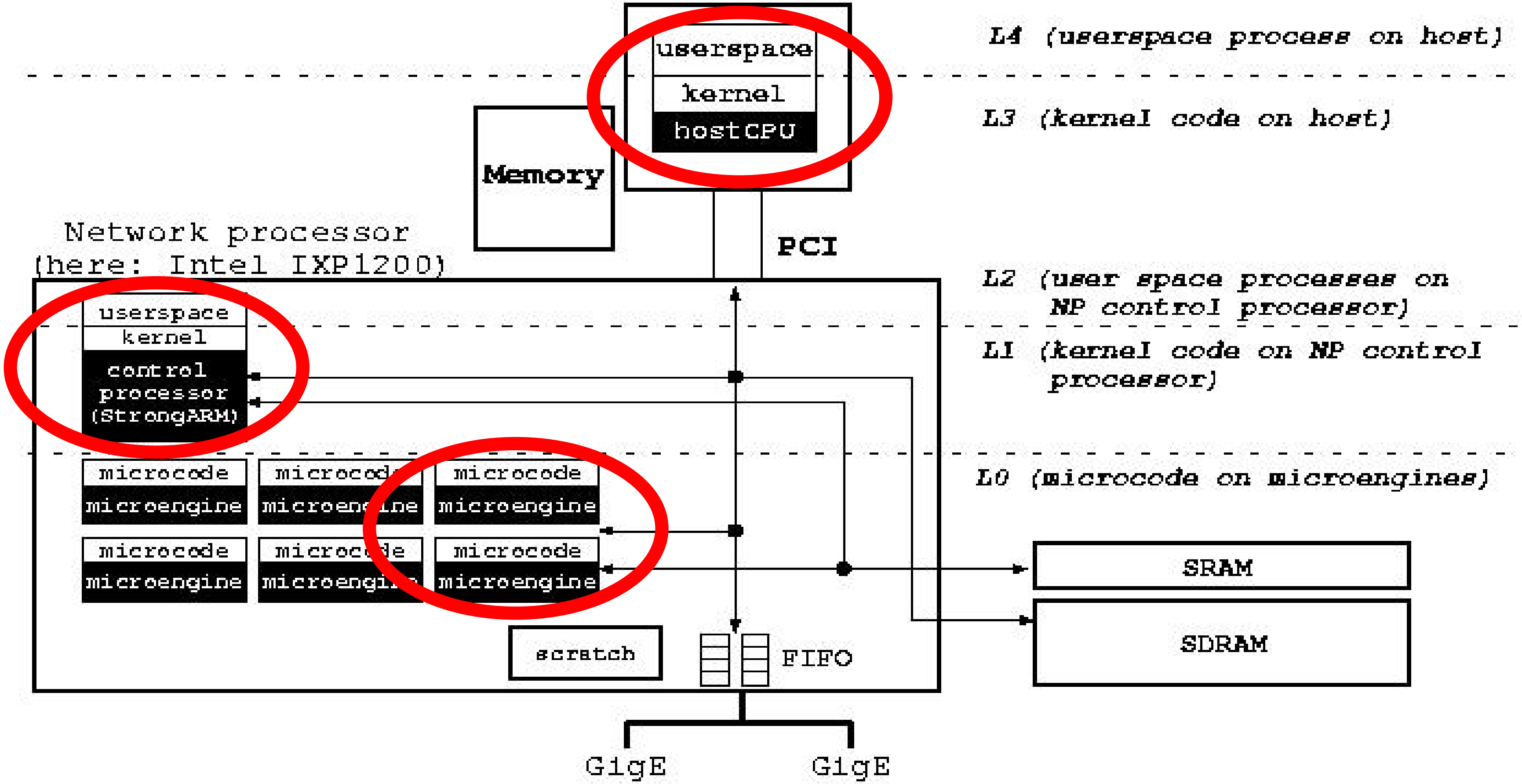
OKE sub-projects

- OKE Corral (“OKE, Click and a dash of active networks”)
- Diet OKE (“network processors”)

OKE Corral:



HOKE-POKE: applying OKE concepts to multiple levels



push monitoring functionality to the microengines

- counters
- filters

Conclusions

Advantages

- OKE may form a basis for resource control even when there are multiple, mutually mistrusting parties
- OKE provides resource control if required, while not incurring overhead, if not
- OKE authorisation procedures can be applied throughout SCAMPI
- OKE overhead can be very small indeed

Disadvantages

- runtime resource control comes at a runtime cost
- writing ESC is complex

In SCAMPI

-
  ```
  // authorise any operation
  scampi_set_authorisation_creds(priv, pub, creds)
  ```

-
  ```
  // load code (OKE or other) anywhere
  scampi_load_code (id, type, location, param)
  ```

-
  ```
  scampi_unload_code (id)
  ```

Diet OKE

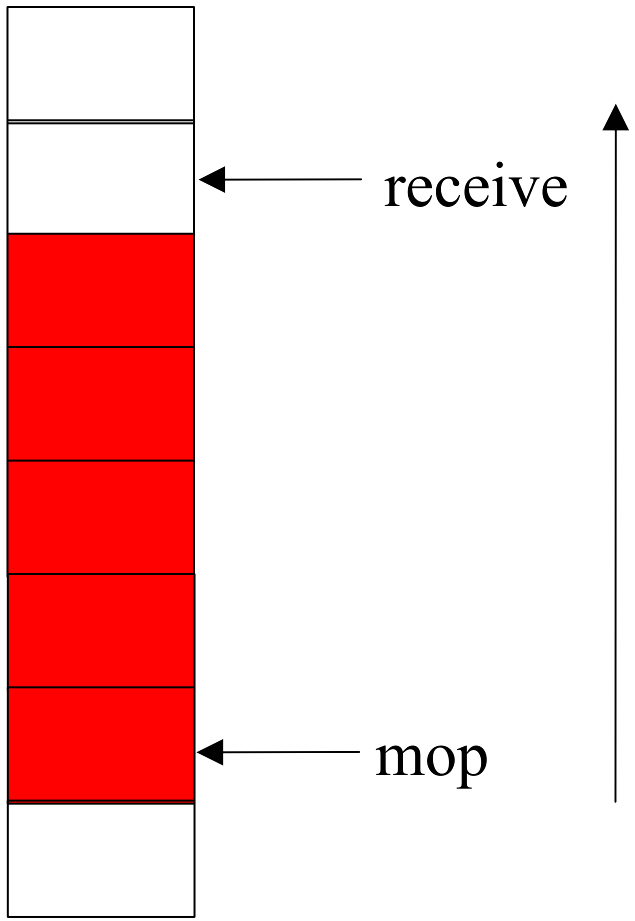
  - multiple application simultaneous access to microengines
  - application granularity: microengine
- killing applications running on MEs
  - by control processor (StrongARM)
- packet access
  - prefiltering
  - locked fields (compile time)
- memory access
  - static memory allocation + protection in API
  - bounds checking at runtime
- stack is not a problem

Diet OKE

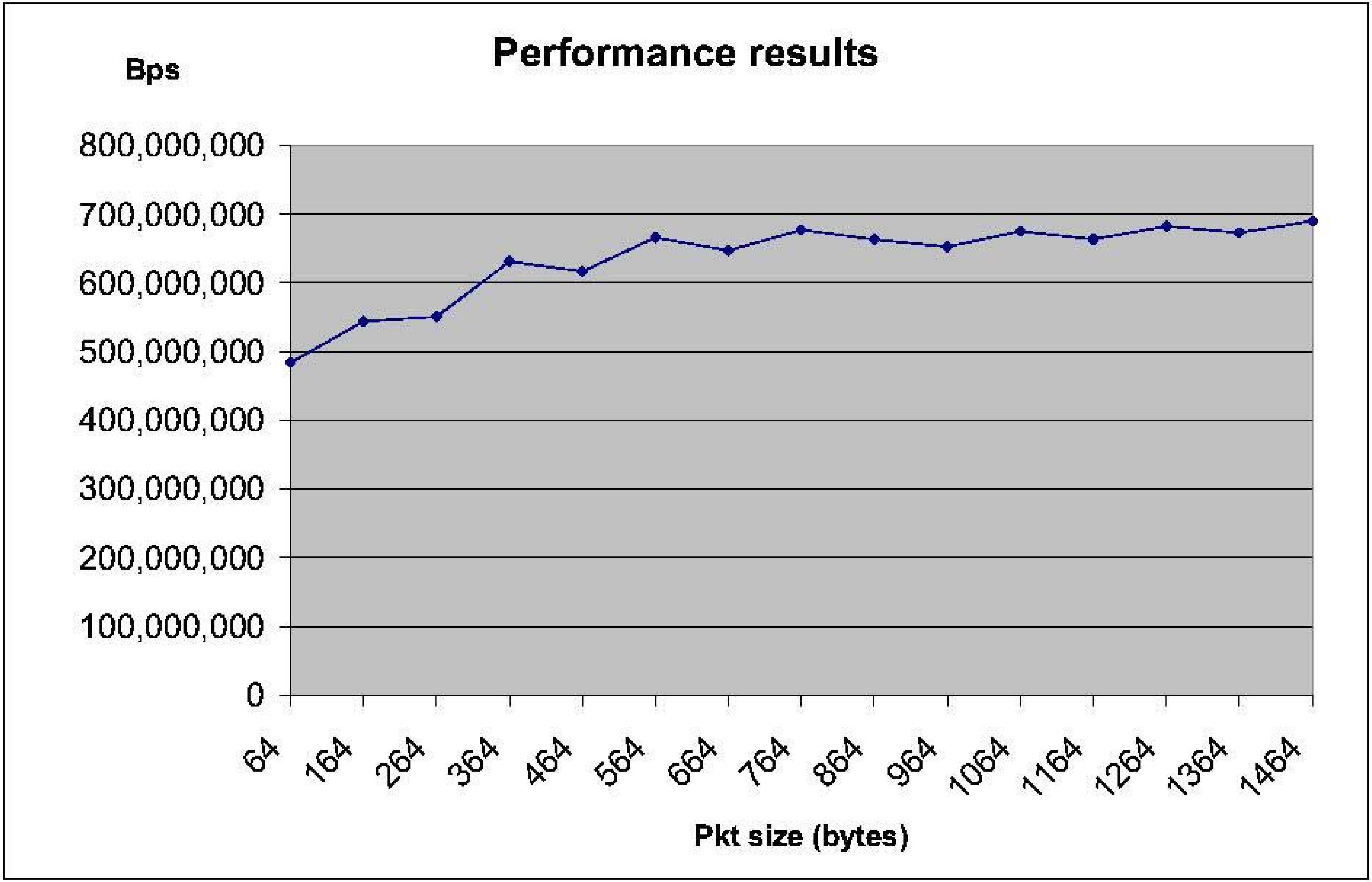
- application framework: granularity = microengine
- 1 ME for packet reception
- 5 MEs for applications
- target: network monitors
- third parties can “plugin” their own functions
- 4 threads

Diet OKE

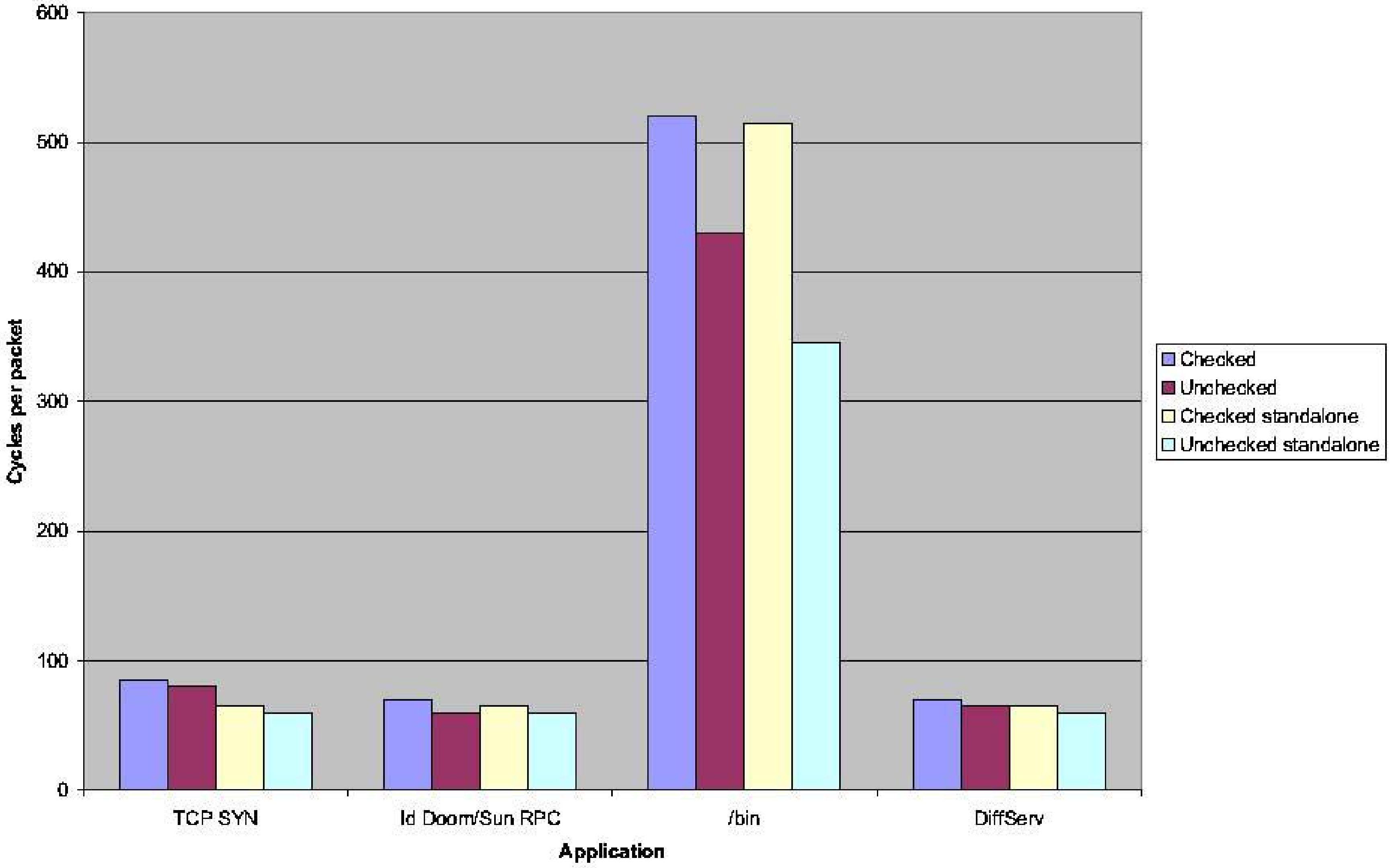
- packets received in circular buffer structure in SRAM + SDRAM
- applications can read packets whenever data is available
- fields:
  - Count
  - flags: W, R*, Done*
- mopping mechanism....



Diet OKE Throughput



Diet OKE Applications



Garbage collector

- mark and sweep, assumes strong typing
- automatically generated code based on whole program analysis (compiler front-end)
- compiler detects which types can be allocated by module
  - by enumerating types of new and malloc
  - generates marking functions for each type
  - defines how to scan mem block of specific type for pointers
  - contains call to GC for every pointer in type
  - mem allocation calls: pass * marking function to mem subsystem
  - GC time: call marking functions for each block