Anatomy of a Microgrid OS

AKA, How does the 800lb gorilla yawn in the kitchen when nobody's watching.

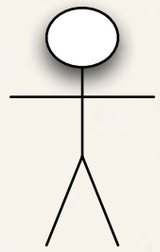
January 26th, 2011

"Zonder verhaal zijn feiten sprakeloos" — K vd Toorn

Your mission, if you accept it: run programs on the Microgrid

- Context: Microgrid users vs. implementers — theory and practice
- Bridging the gap — back to basics, strategy and design overview
- Programming interfaces, components and run-time interactions
- Future research and directions

Context — from the hardware perspective



Hardware guy

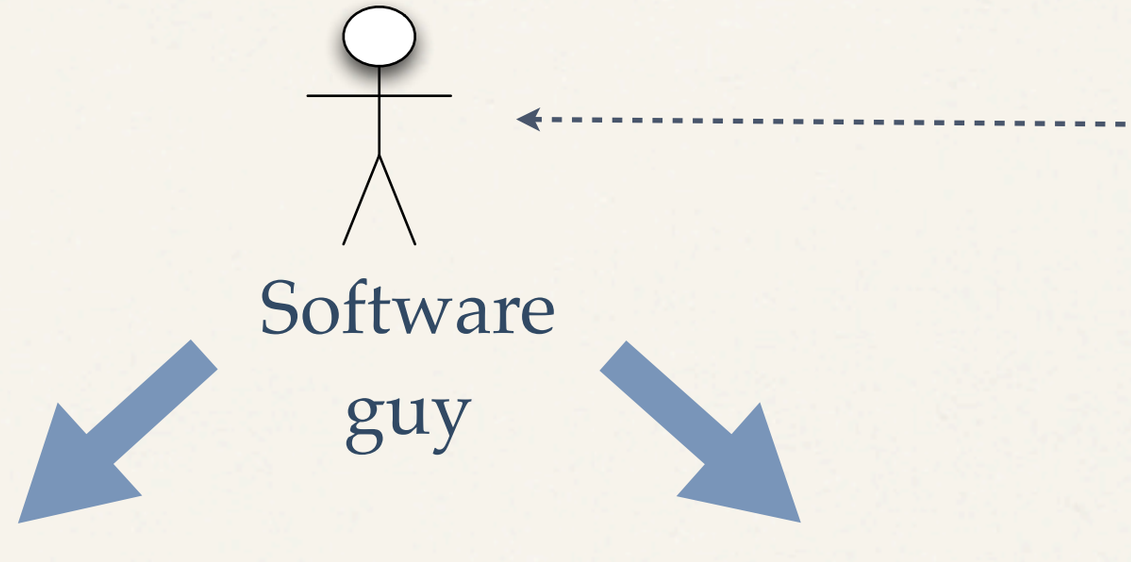
Mike

Chris Jony

μT-LEON3 crew

- **Provides**:
  - Microarchitecture + device specs
  - Integration parameters
  - Low-level bootstrap code
- **Expects**:
  - Unstructured static binary code + data
  - Questions about low-level behavior

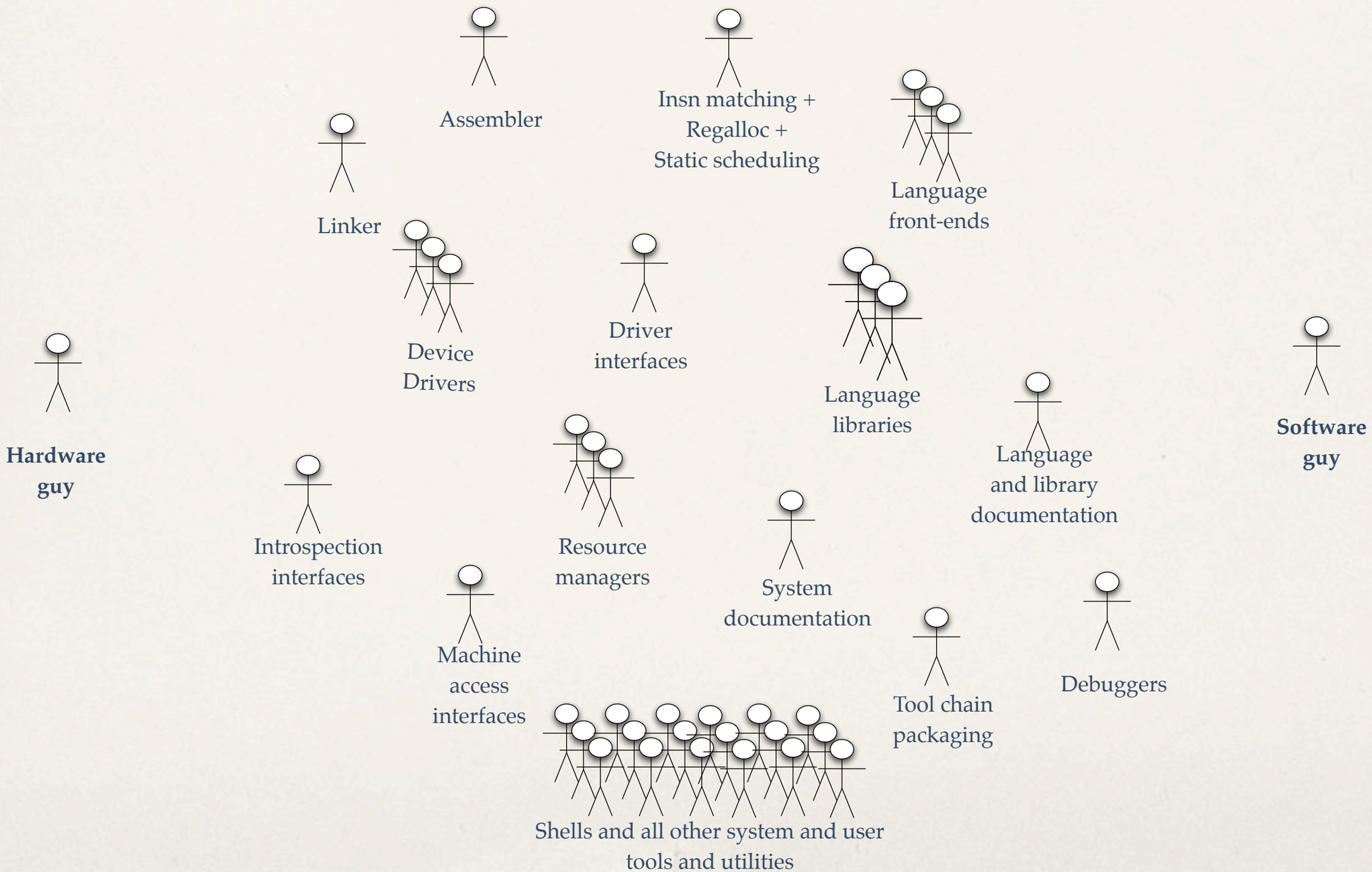
Context — from the software perspective



- **Provides**:
  - Source code, optionally using language extensions
  - Informal description of requirements

- **Expects**:
  - Language and library specs
  - (Some) backward compatibility
  - Tool usage and interfaces
  - Troubleshooting methods

… and the crowd “in between”



It can’t be that complicated?

```
int main(void)
{
   printf (“pi = %f\n”, 3.14);
   return 0;
}
```

It can’t be that complicated?

```
int main(void)
{
   printf (“pi = %f\n”, 3.14);
   return 0;
}
```

Who / what calls main()?

It can’t be that complicated?

```
int main(void)
{
   printf (“pi = %f\n”, 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

vfprintf(uio, "...", ...);

What is "vfprintf"?

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

vfprintf(uio, "...", ...);

What is "vfprintf"?

strnlen() dtoa() malloc() free()

...

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

vfprintf(uio, "...", ...);

What is "vfprintf"?

strnlen() dtoa() malloc() free()

...

...

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

vfprintf(uio, "...", ...);

What is "vfprintf"?

strnlen() dtoa() malloc() free()

...

...

...

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

vfprintf(uio, "...", ...);

What is "vfprintf"?

strnlen() dtoa() malloc() free()

...

...

...

__sfvwrite(uio, ...)

It can't be that complicated?

```
int main(void)
{
   printf ("pi = %f\n", 3.14);
   return 0;
}
```

Who / what calls main()?

What does printf() do?

fprintf(stdout, "...", ...);

What is "stdout"? What is "fprintf"?

FILE *stdout = ...;

What is "FILE"?

FILE { uio; flags; }

What is "uio"?...

vfprintf(uio, "...", ...);

What is "vfprintf"?

strnlen() dtoa() malloc() free()

...

...

...

__sfvwrite(uio, ...)

write(fd, ...)

Ah, here is our output. Or is it?

The Microgrid "vision"

"Today"

Multiple sequential applications

Coarse time sharing software scheduler

Single processor

Extend languages

Shift responsibility

Multi-granularity concurrent code

Hardware concurrency & scheduling

Many cores

"Tomorrow"

Microgrid

The Microgrid reality

Today:

Multi-tasked applications	Time and space sharing schedulers	Multiple cores
Language inter-operability	Standard system services	Memory hierarchies
Portability, compatibility	Device drivers	I/O devices

Expectations **Standard practices**

What have we done?
What do we do now?

Hardware abstraction

- *What*: Common software interfaces to access different hardware.
- *Why*: Needed even for single applications, as soon as *hardware portability* is required — of interest mainly for users.
- *How*: The common interface is exposed (and documented) in at least one programming language. A driver is an implementation of the common interface for a specific device.
- *Who*: OS designers together with hardware providers.
- *When*: Interfaces early during OS design, drivers before/during deployment. Some iteration afterwards.
- *Trade-off*: complex driver interfaces ↔ simple interfaces
  flexibility for future hardware designs ↔ faster adoption, time to market

System services

- *What*: Primarily coordinate access to shared physical resources.
  (Secondarily, provide state-of-the-art algorithms for common tasks, including access to shared conceptual resources, e.g. mutexes)

- *Why*: Needed even for a single hardware design, as soon as software is "too complex" — of interest mainly to programmers.

- *How*: A protocol to ensure the *state* of shared resources; interfaces are often irrelevant.

- *Who*: OS designers together with system architects.

- *When*: Typically, incrementally during the lifetime of software.

- *Trade-off*: complex software protocols ↔ simple protocols
  more control for programmers ↔ more opportunities for system optimization

Operating systems and languages

- Applications provide a purpose; hardware provides a means; OS and language implementations enable programmers.

- A microarchitecture establishes a real physical abstraction, between hardware and software

- In contrast, the distinctions between OS and language abstractions are mainly social and organisational; their implementers share the same audience (programmers)

- These distinctions can be enforced physically — via memory protection, process isolation, "trusted computing" etc. — but this is *not strictly necessary* to program and use computers!

Operating systems and languages

- Applications provide a purpose; hardware provides a means; OS and language implementations enable programmers.

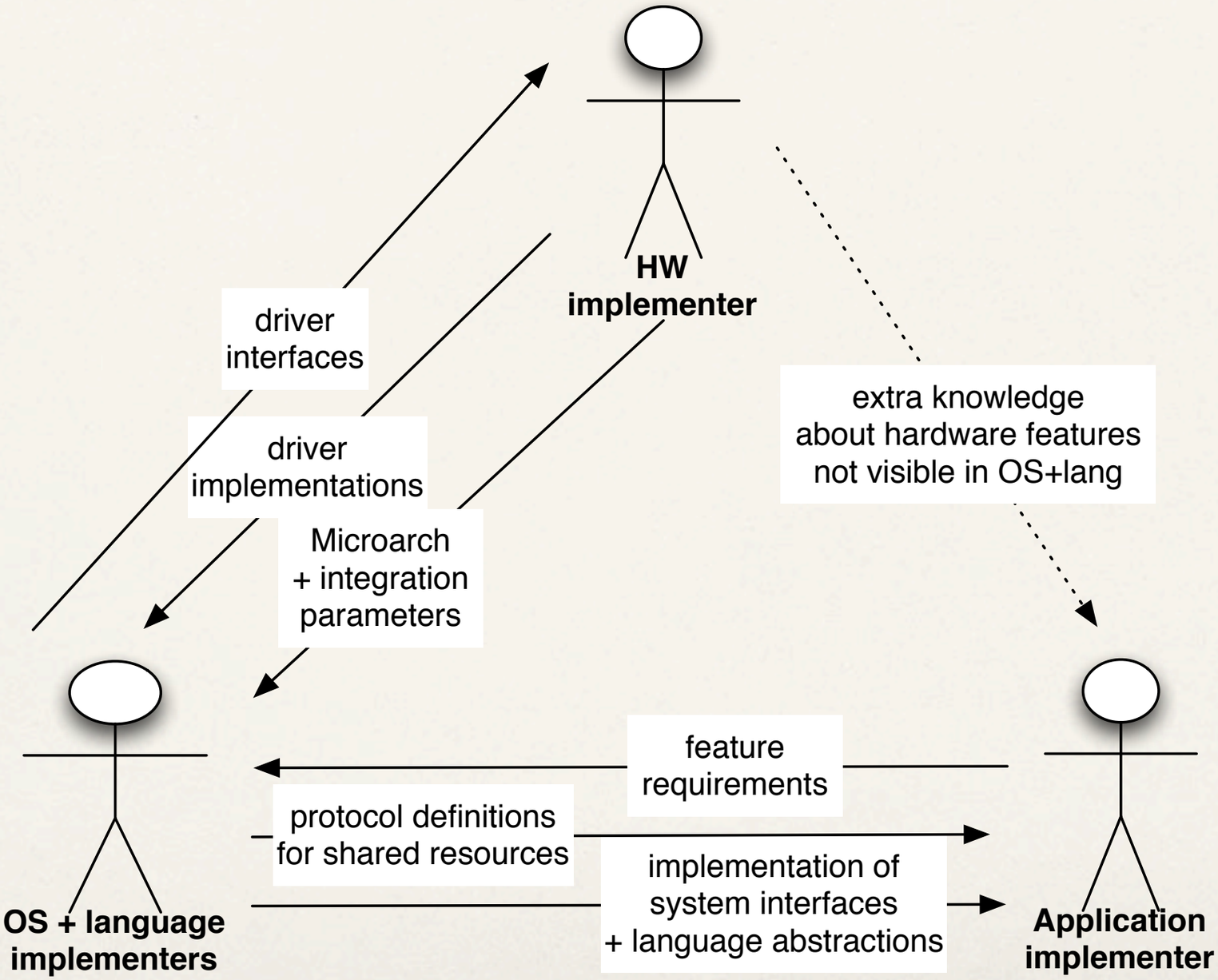
- A microarchitecture establishes a real physical abstraction, between hardware and software

- In contrast, the di… …ons are mainly soci… same audienc…

- These distinctio… protection, proces… but this is *not strictly n…* …ers!

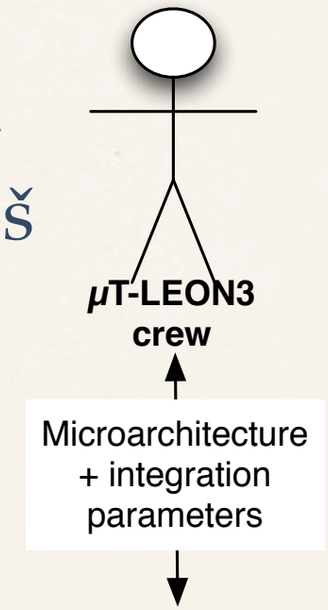
Consider: *"The C library is considered part of the operating system on Unix-like systems. Unix-like operating systems and Unix programs generally cannot function if the C library is erased. By contrast, on Microsoft Windows, the core system dynamic libraries (DLLs) do not provide an implementation of the C standard library; this is provided by each compiler individually.*

Synthetic actors and their responsibilities



Better yet — the Microgrid ecosystem is small

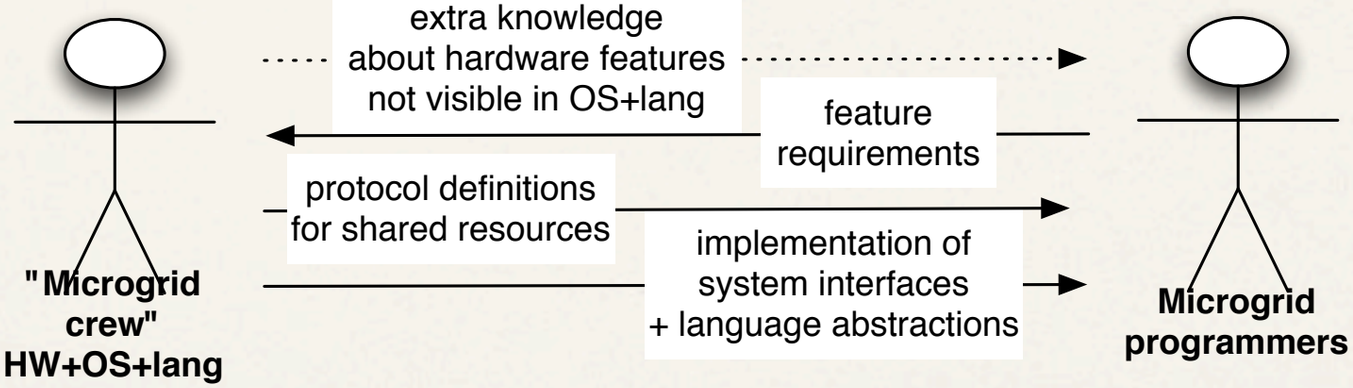
Martin
Jara Leoš



NWO Microgrids
ADVANCE
FADE(?)
Apple-CORE
MODERN

"SAC/μTC team"

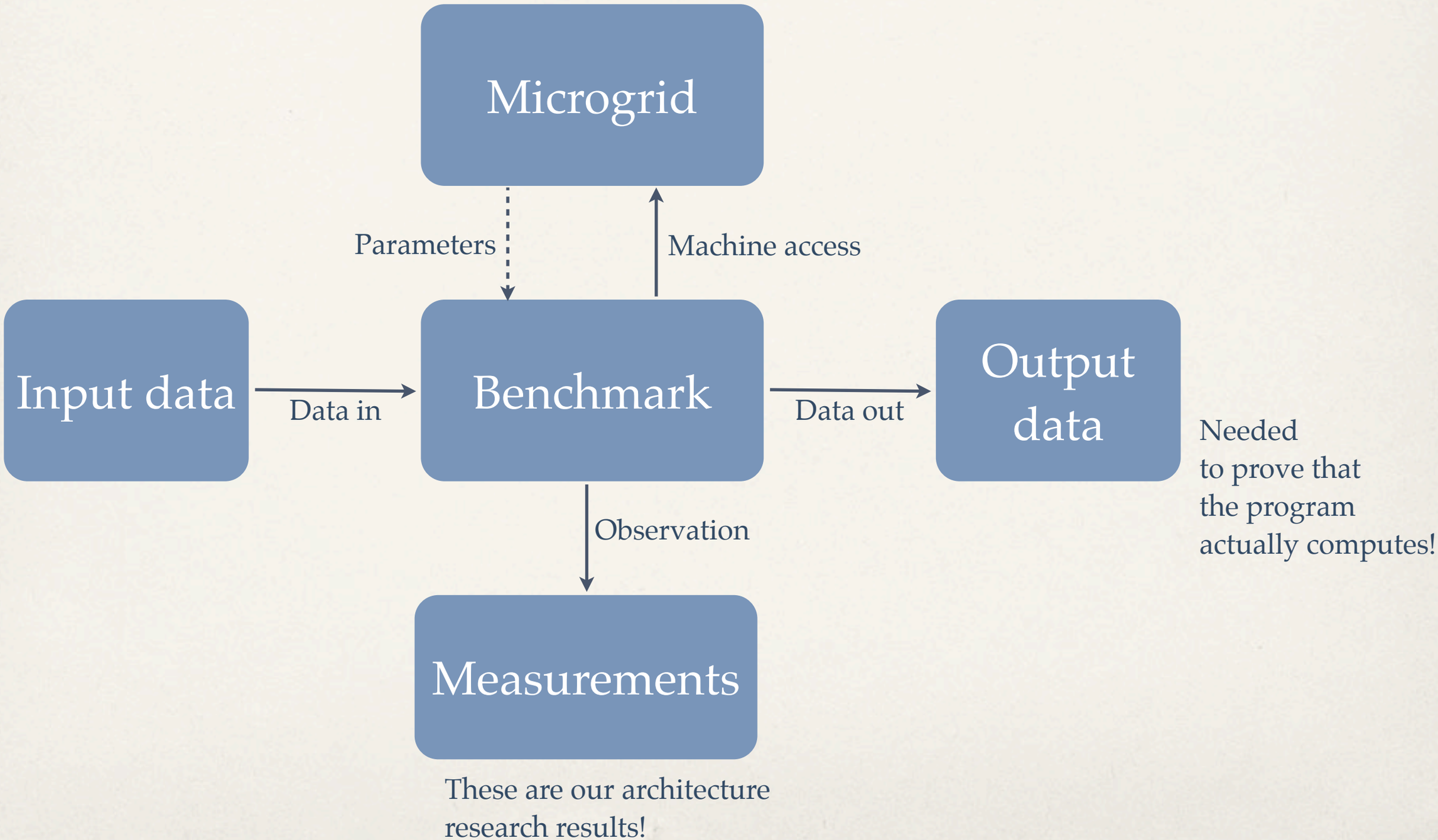
Mike
Yang
Michiel
Michael
Thomas
Chris
Jony
Kostas



Carl
Stephan
Nils
Bodo
Clemens
Dimitris (**UoI**)
Andrei
Irfan
Yang

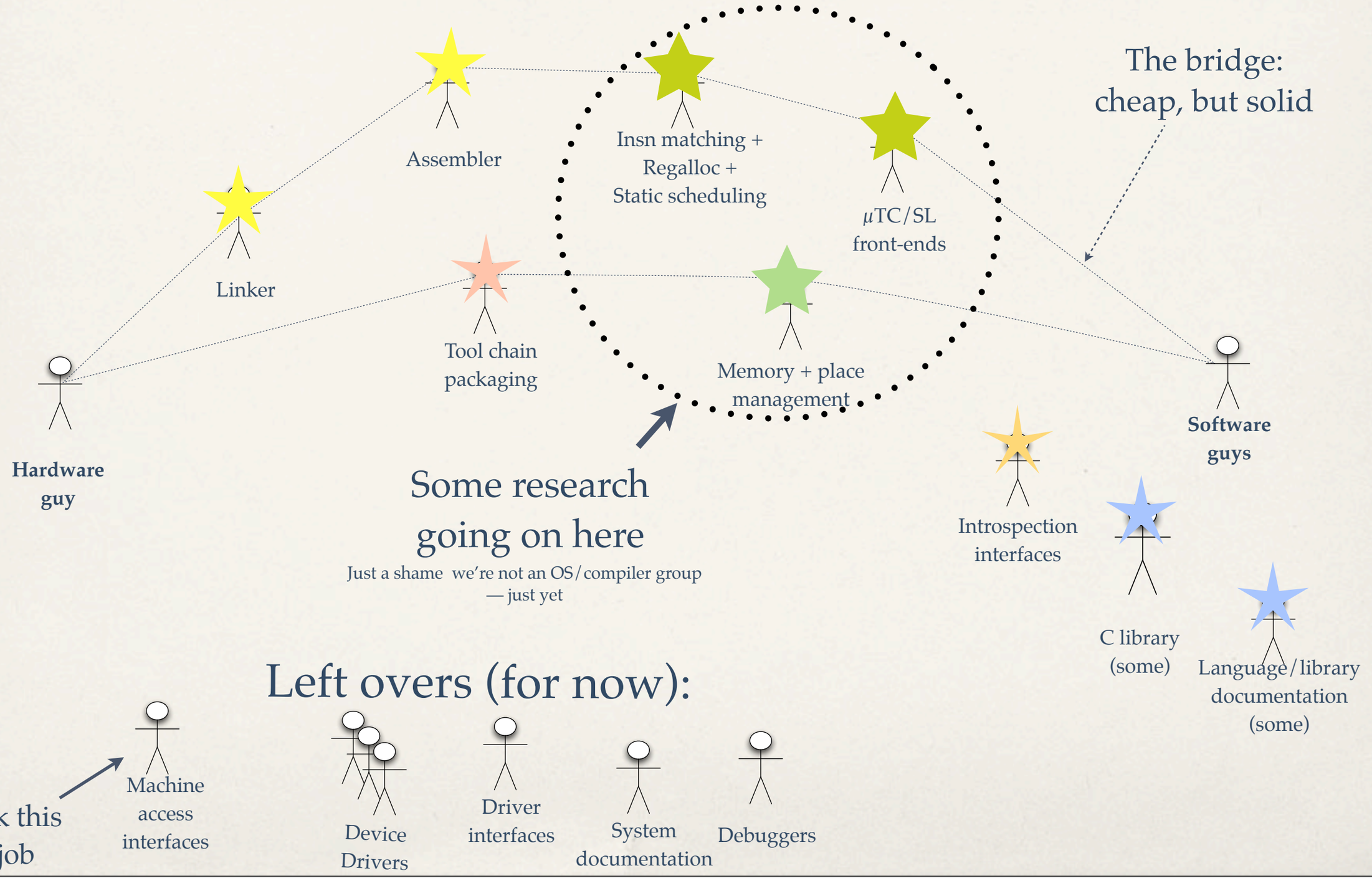
MSc students

The smallest feature set required to demonstrate the MG



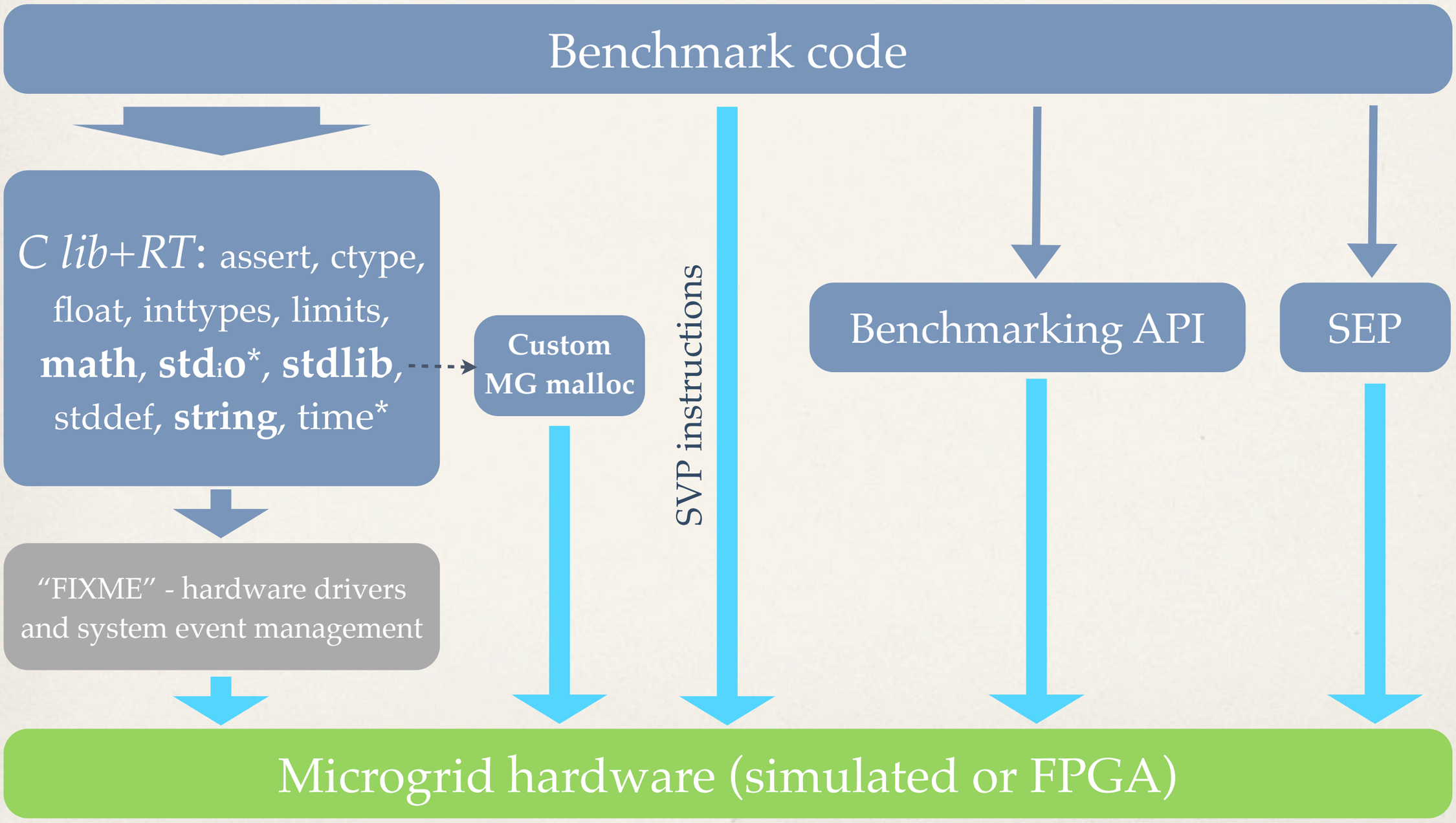
A first bridge: Apple-CORE

A first bridge: Apple-CORE



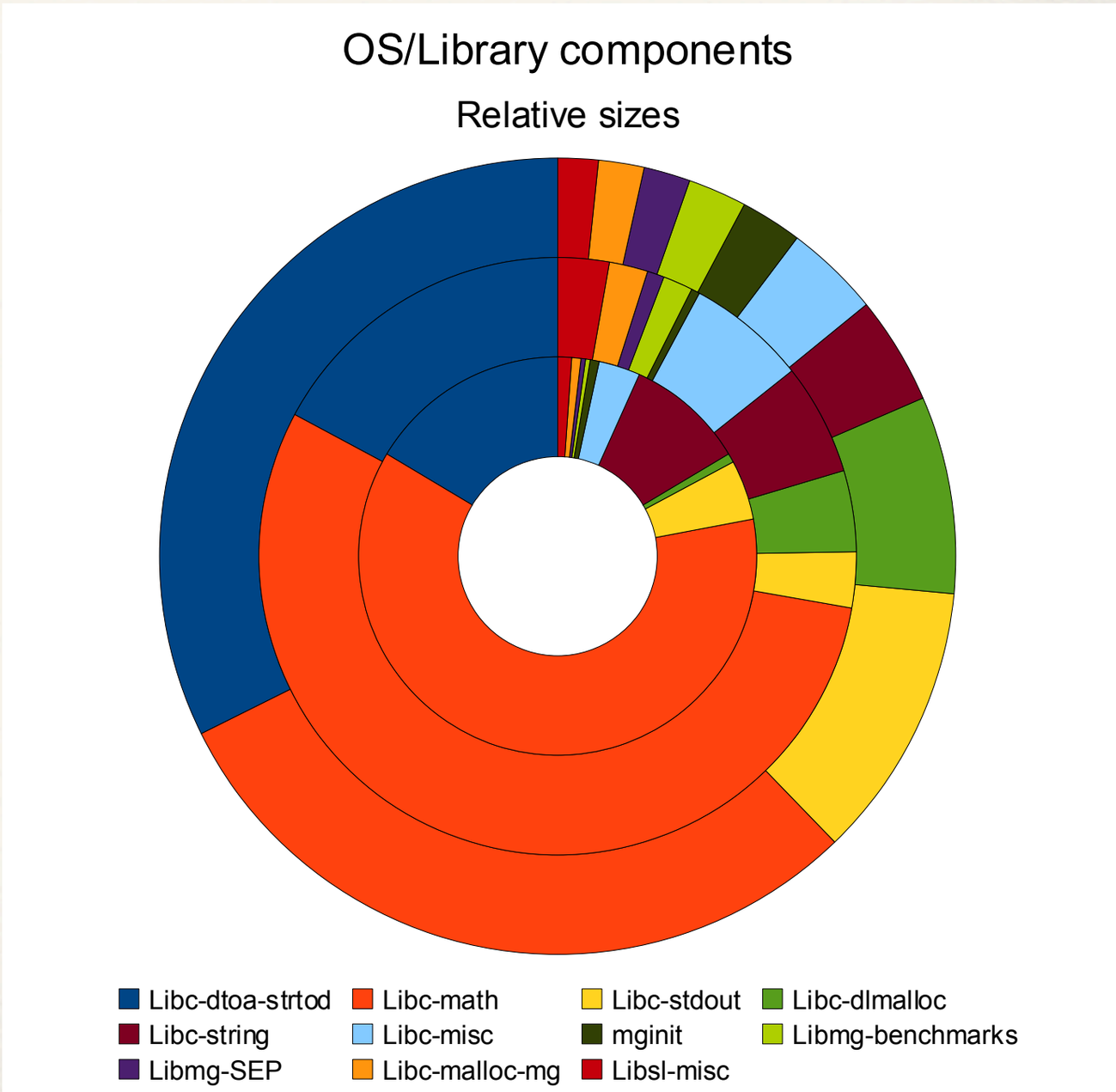
A first bridge: Apple-CORE

MGOS: Software interfaces



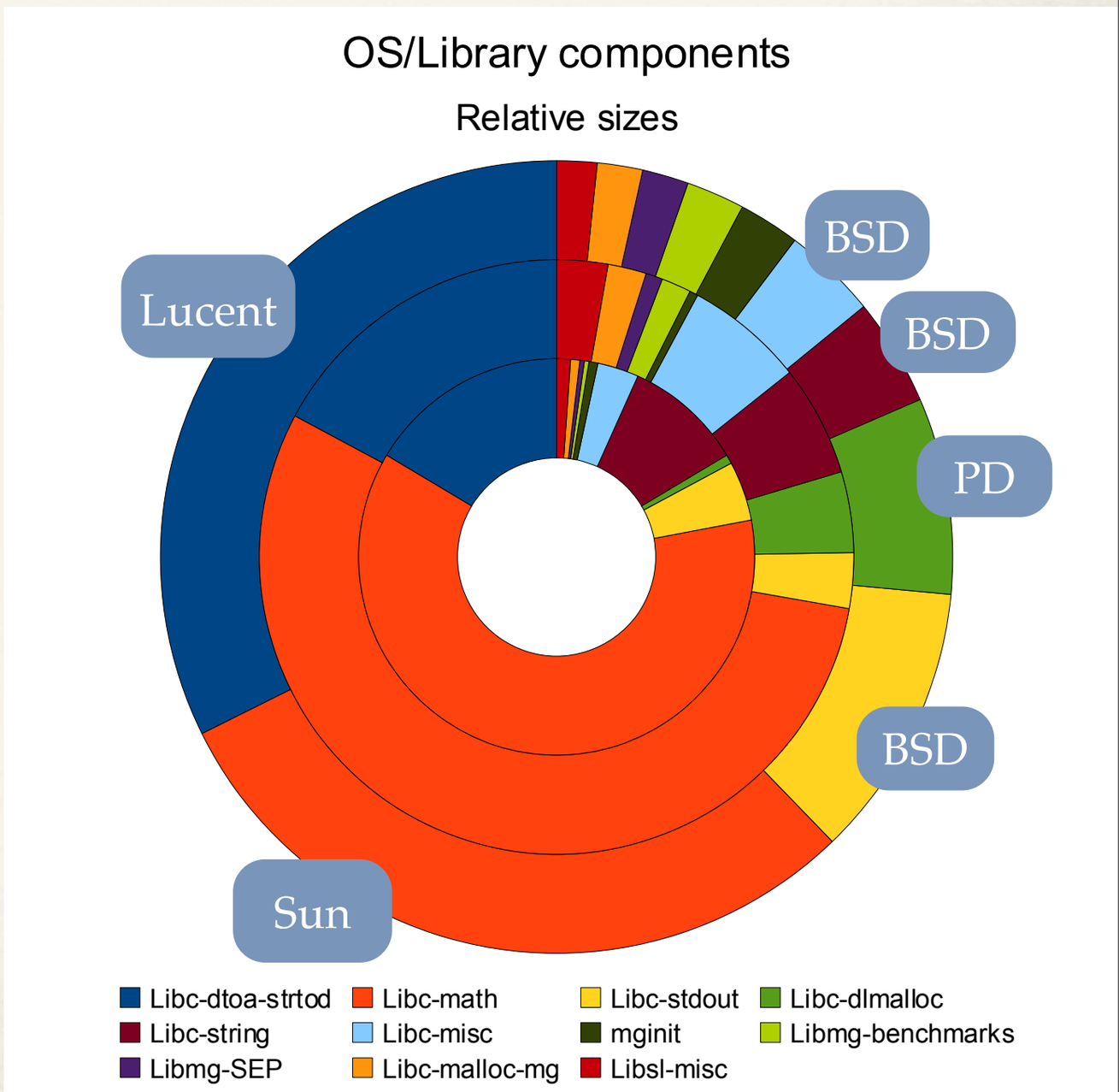
Inside the MGOS

Component	Objects	Functions	Code words
Libc-dtoa-strtod	44	74	14528
Libc-math	165	238	13424
Libc-stdout	13	13	5040
Libc-dlmalloc	2	19	3600
Libc-string	26	26	1968
Libc-misc	9	28	1728
mginit	2	2	1120
Libmg-benchmarks	1	7	1072
Libmg-SEP	1	4	848
Libc-malloc-mg	2	9	832
Libsl-misc	3	12	736



Inside the MGOS

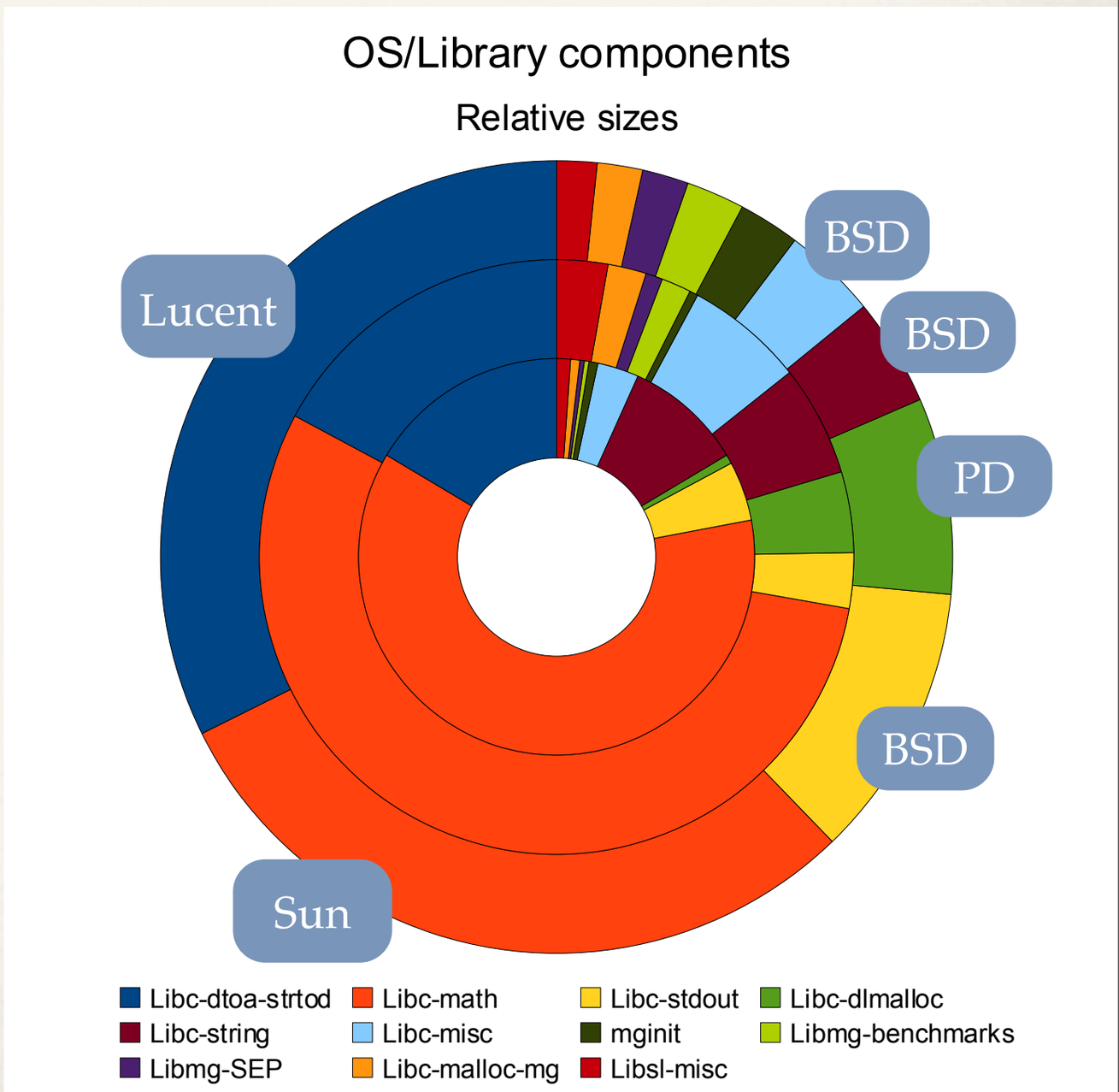
Component	Objects	Functions	Code words
Libc-dtoa-strtod	44	74	14528
Libc-math	165	238	13424
Libc-stdout	13	13	5040
Libc-dlmalloc	2	19	3600
Libc-string	26	26	1968
Libc-misc	9	28	1728
mginit	2	2	1120
Libmg-benchmarks	1	7	1072
Libmg-SEP	1	4	848
Libc-malloc-mg	2	9	832
Libsl-misc	3	12	736



Many copyright holders...

Inside the MGOS

Component	Objects	Functions	Code words
Libc-dtoa-strtod	44	74	14528
Libc-math	165	238	13424
Libc-stdout	13	13	5040
Libc-dlmalloc	2	19	3600
Libc-string	26	26	1968
Libc-misc	9	28	1728
mginit	2	2	1120
Libmg-benchmarks	1	7	1072
Libmg-SEP	1	4	848
Libc-malloc-mg	2	9	832
Libsl-misc	3	12	736

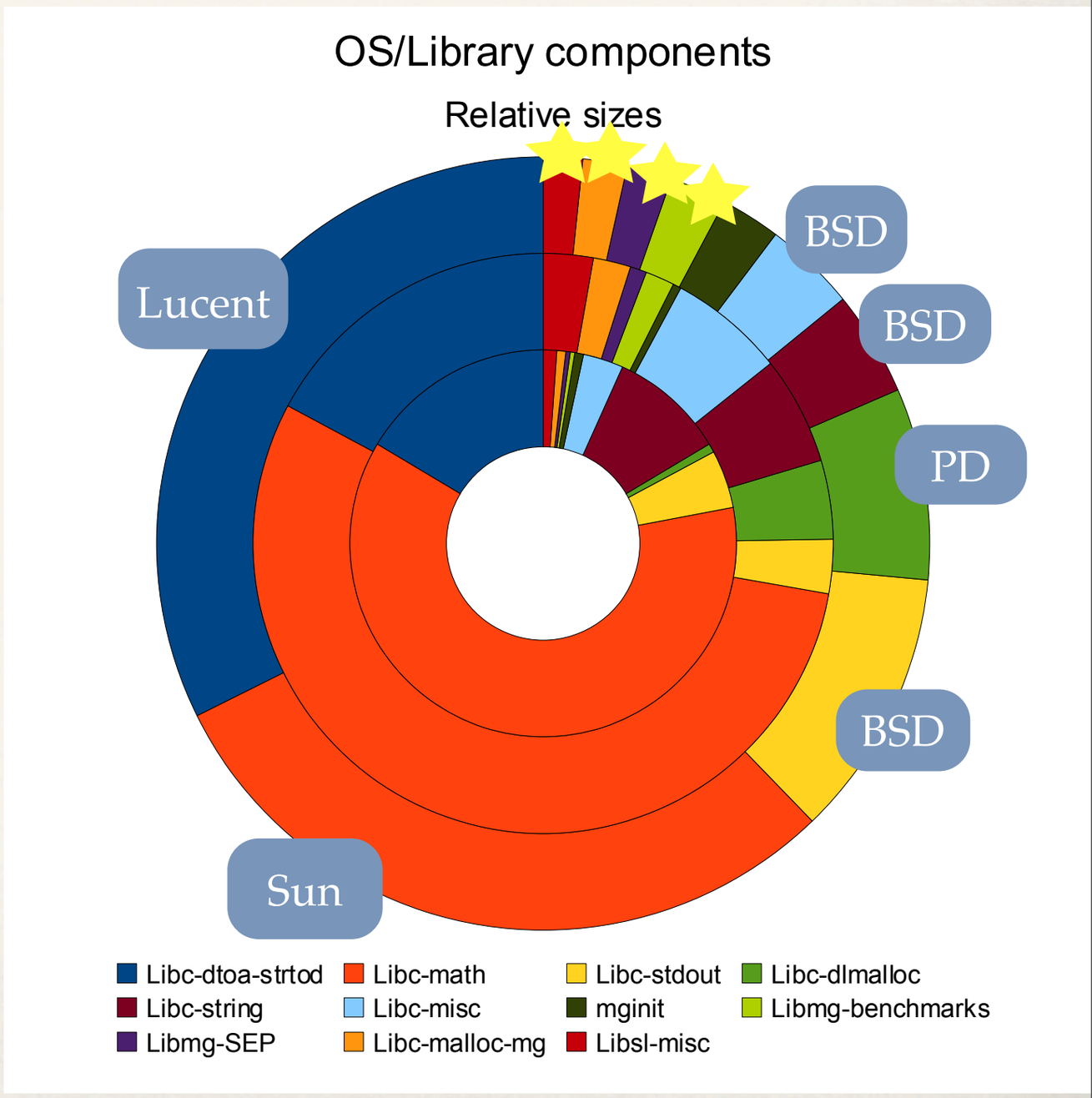


Many copyright holders...

... but only one import tree: FreeBSD

Inside the MGOS

Component	Objects	Functions	Code words
Libc-dtoa-strtod	44	74	14528
Libc-math	165	238	13424
Libc-stdout	13	13	5040
Libc-dlmalloc	2	19	3600
Libc-string	26	26	1968
Libc-misc	9	28	1728
mginit	2	2	1120
Libmg-benchmarks	1	7	1072
Libmg-SEP	1	4	848
Libc-malloc-mg	2	9	832
Libsl-misc	3	12	736

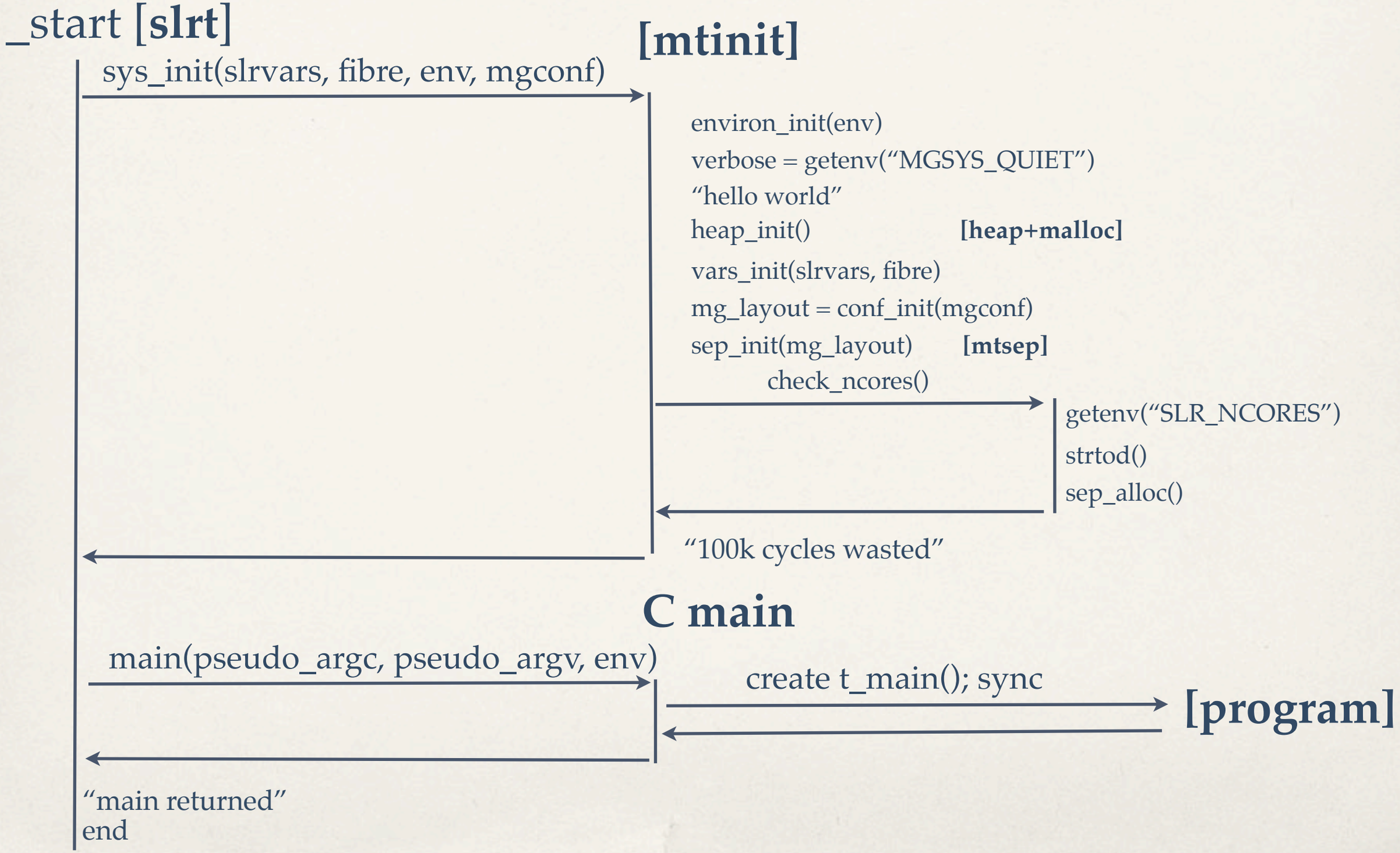


Many copyright holders...

... but only one import tree: FreeBSD

And comparatively little MG-specific code!

Bootstrap — initial call graph



Strategy and future research

- Current MGOS can run non-trivial code, but only single programs
  e.g. the full Livermore FORTRAN benchmark from 1983 (with f2c)

- Next step: run multiple applications side by side. Larger data.

- Requires: process management. virtualized I/O. file access.
  However: we do not want to rewrite a VIO/VFS layer ourselves. Need to reuse!

- Maybe look again at porting an existing OS?
  This in turn requires (bare minimum): Thread preemption for at least 1 context. A timer interrupt. An I/O device. With this we can probably get μcLinux. Cheap route: add a single-threaded gen-purpose core on the Microgrid with a common ISA.
  Add virtual addressing for the preemptible context and we can get NetBSD.

- Alternative next step: emulate POSIX threads and/or GOMP and/or MPI. Then evaluate performance of existing single application, multithreaded code.

Thank you!