

TAL — Feature Model

Operating-System Engineering

Thread Abstraction Layer — TAL

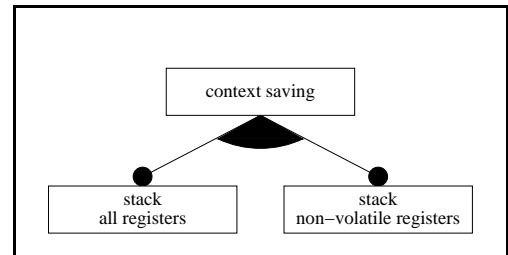
- design fundamental abstractions related to the following topics:
 - control flows 2
 - low-level scheduling 10
 - CPU management 12
 - compiler dependencies 17
- aim at providing a *minimal subset of thread functions*

Context Saving

- a thread's context is made of the contents of its processor registers
 - its size depends on the CPU and the compiler/programming language
 - it needs to be temporarily saved during phases of thread inactivity

- stack-based context-saving shall come true:

1. stack all CPU registers
2. stack non-volatile CPU registers
3. policy 1. as well as policy 2.

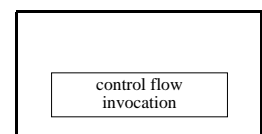


- the system description must contain at least one of these variants: *or-feature*

Control Flow Invocation

- provide a thread concept basing on an *asymmetric invocation* mechanism
 - allow the spawning of a thread “inline” at (almost) any point of execution
 - the spawned thread preempts the spawning thread, it runs until completion
 - upon completion the spawned thread terminates and releases control

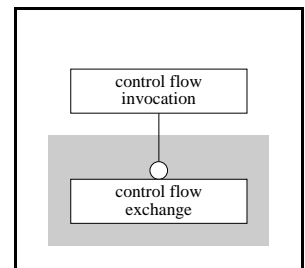
- inherit the (total) processor context to the spawned thread
 - the only private piece of context of the thread is its stack
 - thus thread invocation only implies exchanging the stack



- upon completion, the processor context is (re-)inherited to the resumed thread

Control Flow Exchange

- extend the thread concept by means of a *symmetric invocation* mechanism
 - exchanging the flows of control happens in a coroutine-like fashion
 - a control-releasing thread remembers its resumption point (return address)
 - the thread's resumption point will be saved on the thread's run-time stack
- the threads all share the same processor context (→ p. 3)
 - thus thread resumption only implies a stack change
- resumption happens “inline”, at any point of execution

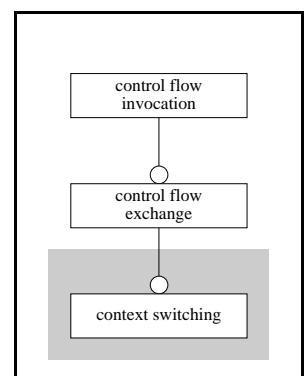


Context Switching

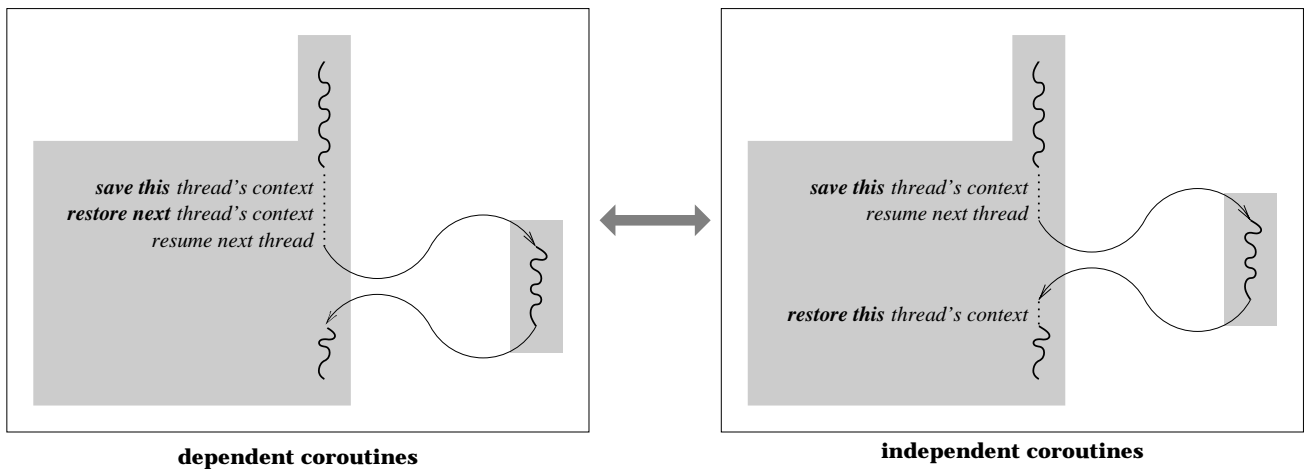
- extend the thread concept by a *coroutine* style of context switching
 - context saving (restoring) happens before (after) a resumption point
- provide thread switching in two ways:

$\left\{ \begin{array}{l} \text{inline} \\ \text{procedural} \end{array} \right\}$ taking care of the $\left\{ \begin{array}{l} \text{complete} \\ \text{non-volatile} \end{array} \right\}$ context

- requires the context-saving feature (→ p. 2)
- threads need to save/restore their contexts by themselves

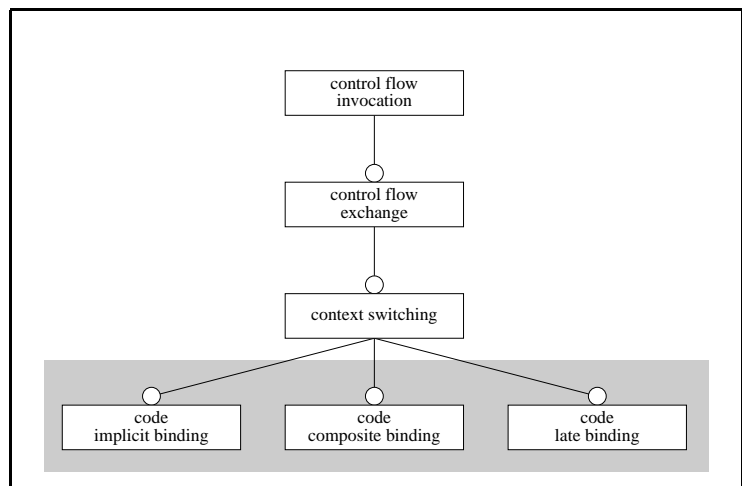


Types of Coroutines



Code Binding

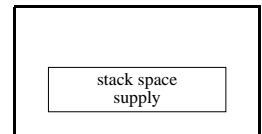
- extend the thread concept to support the binding of *user-defined code*
- three bindings are nice to have:
 1. any sort of code fragment
 2. pointer to function
 3. specialized (virtual) function
- any or none of them is valid
 - *optional features*
- aims at “user-friendliness”



Stack Space Supply

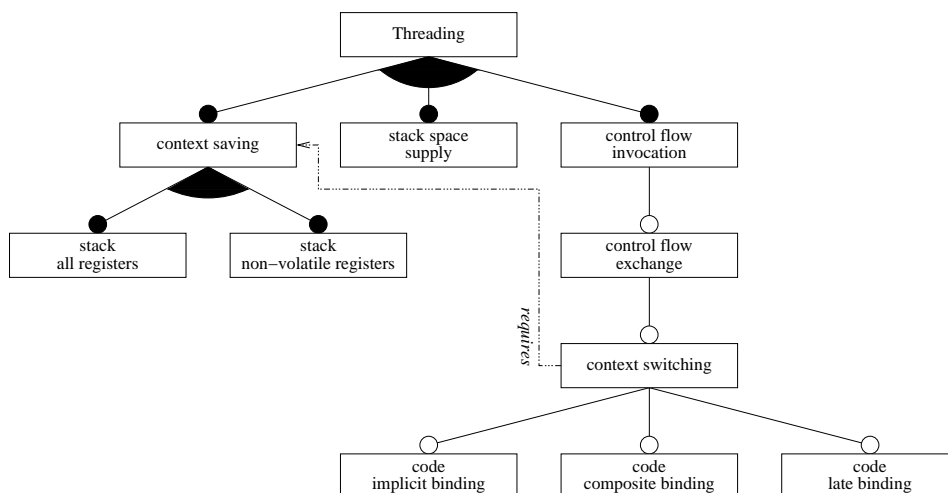
- provide a concept for *compile-time stack-space allocation* for a thread
 - a specialization of the fundamental thread abstraction (→ p. 3)
 - a (programming-language) representation as a template class, e.g.

- purpose is to somehow provide means for “user-friendliness”
 - an abstraction that should be optional, nevertheless



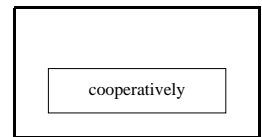
- the feature is nice to have, yet it's non-functional and thus could be void

Threading Concept

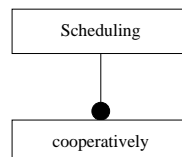


Scheduling

- the threading concept already provides a limited form of *thread scheduling*
 - threads are expected to be scheduled cooperatively, under the user's control
 - * much in a same way as ordinary coroutines
 - they keep on executing until control over the CPU is relinquished explicitly
- there is no (central) system-level thread scheduler
 - thread scheduling is entirely in hand of the application
 - this gives a maximum/minimum of flexibility/support
- CPU protection and other more enhanced policies are subjects for specialization

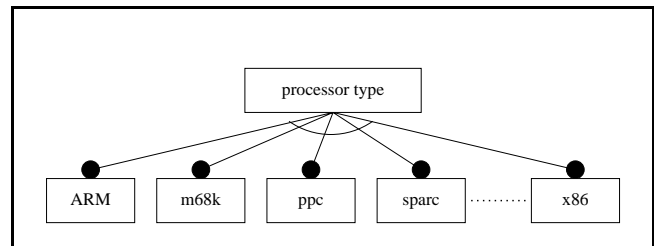


Scheduling Concept



Processor Type

- thread implementations depend on the capabilities of the underlying processor
 - **concrete processor** the thread concept must be implemented from scratch+
 - encompassing assembly-language programming to some extent
 - resulting into a “native implementation” running on the bare machine
 - **abstract processor** the programming language provides a thread concept –
- hardware abstractions are required
 - hiding processor peculiarities
 - enabling portability at user level
- processor as an *alternative feature*



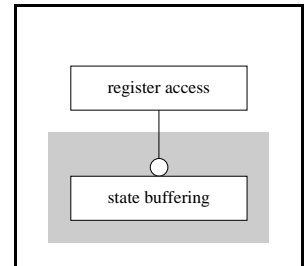
Register Access

- abstractions to access the CPU's registers aid processor-state management
 - reading and writing of registers of the CPU's programming model
 - made feasible e.g. using “inline assembler” and/or `asm()` statements
- a measure to improve the handling of processor-dependent stuff
 - achieving portability is not the purpose at this level of abstraction
- means of operator overloading would be nice to have
 - assignment operators and type casts as in C++, e.g.



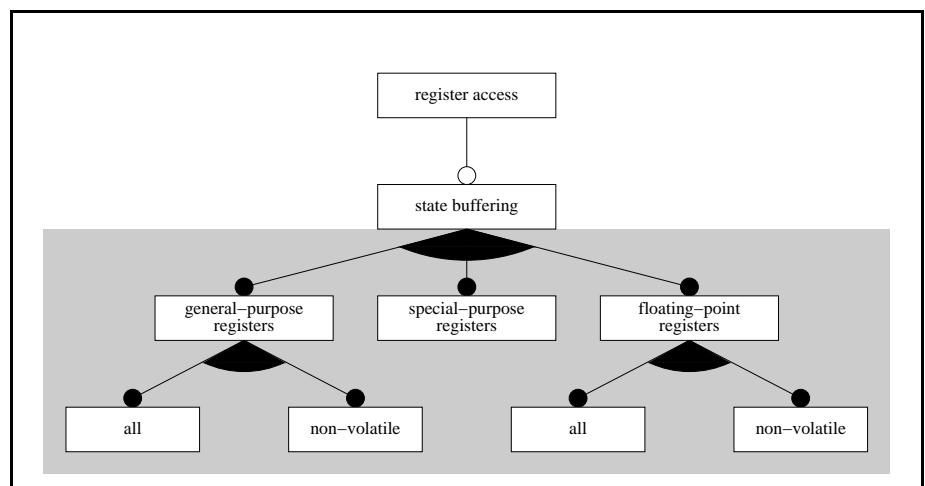
State Buffering

- use register-access functions to save the processor state into a *state buffer*
 - provide primitives to save and restore the processor state
 - the buffer is not implicitly allocated on the stack (→ p. 2)
- hide hardware peculiarities as far as possible
 - the primitives' interface promotes CPU independency
- don't enforce portability: let it go an *optional feature*
 - some users want to explore the CPU by themselves
- but there is the need to distinguish between different sorts of registers . . .

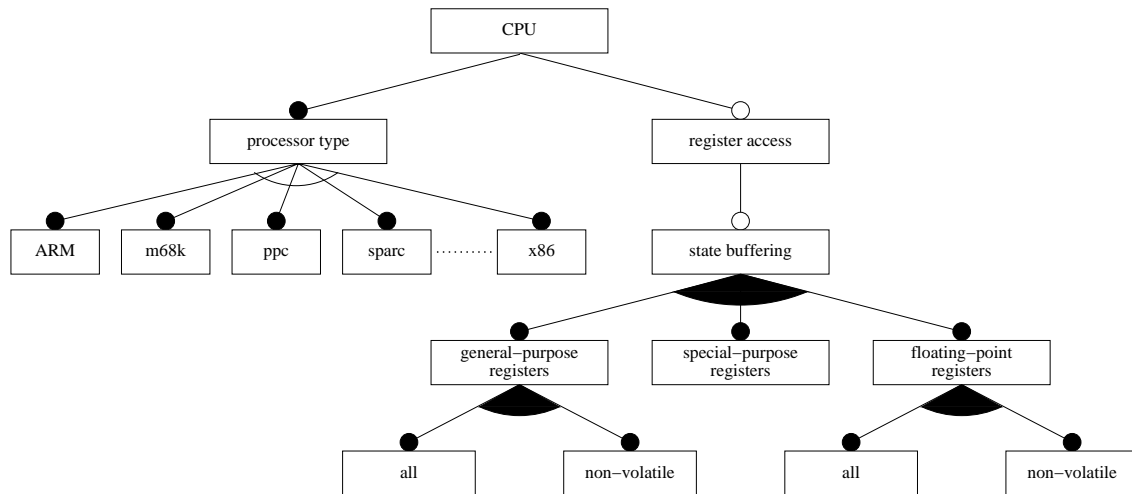


Sorts of Registers

- the register set considered depends on both the concrete and abstract processor
- concrete . . .
 - general purpose
 - special purpose
 - floating-point
- abstract . . .
 - register banks
- sets of *or-features*

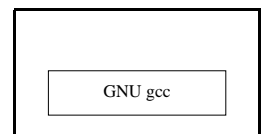


CPU Concept

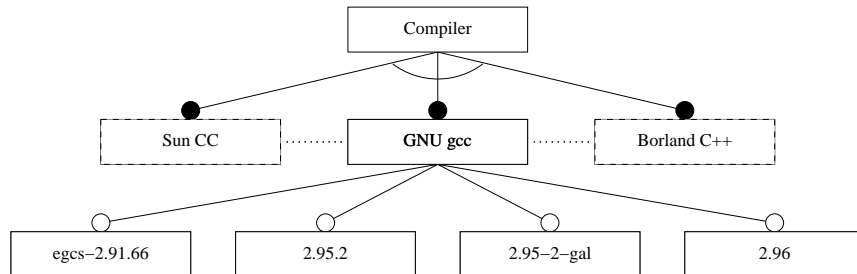


Compiler

- programming language and compiler implement an *abstract processor*
 - sooner or later, both typically undergo more or less major revisions
 - software should differentiate between the various compiler releases
 - * similar to differentiating amongst concrete processors (→ p. 12)
 - portability by far does not only mean to be hardware independent
- programs may depend on the capabilities of that processor
 - e.g. register access using inline assembler (→ p. 13)
- being compiler-independent may be as difficult as being hardware-independent



Compiler Concept



Summary

- system-software design¹ must differentiate between two sorts of requirements:

functional requirements

- threading concept, scheduling concept
- CPU concept (except processor type)

non-functional requirements

- compiler concept
- processor type

- domain-analysis quality largely depends on the analyst's domain experience

¹In general, of course this should hold for any kind of software design.

TAL Concept

